UniTs - University of Trieste

Faculty of Scientific and Data Intensive Computing

Department of mathematics informatics and geosciences

Deep Learning

Lecturer:
Prof. Alessio Ansuini

Author:
Andrea Spinelli

May 27, 2025

This document is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike (CC BY-NC-SA) license. You may share and adapt this material, provided you give appropriate credit, do not use it for commercial purposes, and distribute your contributions under the same license.

github.com/Spina02 andreaspinelli2002@gmail.com

Preface

As a student of Scientific and Data Intensive Computing, I've created these notes while attending the **Deep Learning** course.

The prerequisites of the course are basic knowledge of:

- Linear Algebra (eigenvalue problems, SVD, etc.)
- Mathematical Analysis (multivariate differential calculus, etc.)
- Probability (chain rule, Bayes theorem, etc.)
- Machine Learning (logistic regression, PCA, etc.)
- Programming (Python, Linux Shell, etc.)

While these notes were primarily created for my personal study, they may serve as a valuable resource for fellow students and professionals interested in Deep Learning.

Draft

Contents

1 Introduction

1.1 What is Deep Learning?

Definition: *Deep Learning*

*"**Deep Learning** is constructing networks of parametrized functional modules and training them from examples using gradient-based optimization."*

~ Yann LeCun

In other words, Deep Learning is a collection of tools to build complex modular differentiable functions. These tools are devoid of meaning, it is pointless to discuss what DL can or cannot do. What gives meaning to it is how it is trained and how the data is fed to it.

Deep learning models usually have an architecture that is composed of multiple layers of functions. These functions are called **modules** or **layers**. Each layer is a function that takes an input and produces an output. The output of one layer is the input of the next layer. The output of the last layer is the output of the model.

Practical Applications of Deep Learning

Deep Learning has revolutionized numerous fields by achieving unprecedented performance on complex tasks. In **computer vision**, convolutional neural networks can recognize objects, detect faces, segment images, and even generate realistic images. **Protein structure prediction** has seen remarkable advances with models like AlphaFold, which can accurately predict 3D protein structures from amino acid sequences, fundamentally changing molecular biology and drug discovery. **Speech recognition and synthesis** systems powered by deep learning can transcribe spoken language with near-human accuracy and generate natural-sounding speech, enabling voice assistants and accessibility tools. Other applications include natural language processing (powering chatbots and translation systems), recommendation systems, anomaly detection in cybersecurity, weather forecasting, and autonomous driving. The versatility of deep learning comes from its ability to learn meaningful representations directly from data, reducing the need for manual feature engineering while achieving superior performance across diverse domains.

1.2 Family of linear functions

Let's consider a simple modell

$$y = \Phi_0 + \Phi_1 x$$

This model is a linear function of x with parameters Φ_0 and Φ_1. The model can be represented as a line in the $x - y$ plane. The parameters Φ_0 and Φ_1 determine the slope and the intercept of the line.

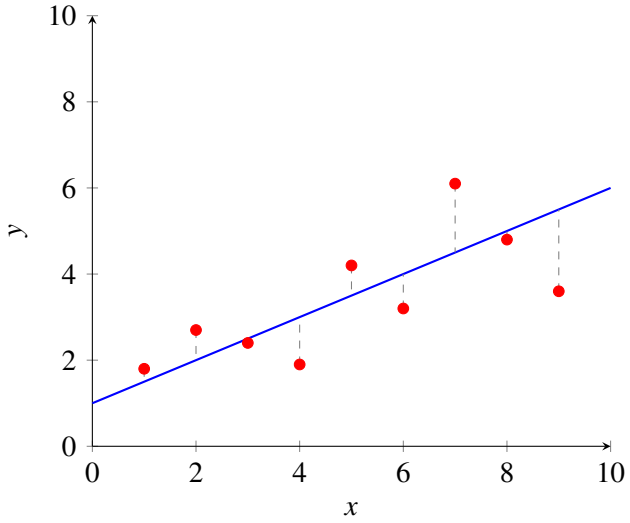


Figure 1.1: Linear model with scatter points and error segments

Loss Function

The loss function for this model is the mean squared error (MSE) between the predicted values and the actual values:

$$L = \frac{1}{N}\sum_{i=1}^{N}(\underbrace{\Phi_0 + \Phi_1 x_i}_{=P_i} - y_i)^2$$

where N is the number of data points, x_i is the i-th input, y_i is the i-th target, and P_i is the predicted value for the i-th data point.

Optimization

The goal of optimization is to find the values of Φ_0 and Φ_1 that minimize the loss function. This is done by computing the gradient of the loss function with respect to the parameters and updating the parameters in the opposite direction of the gradient. The update rule for the parameters is given by:

Let's calculate the gradient of the losso function:

$$\nabla_{\{\Phi\}} L = \left[\frac{\partial L}{\partial \Phi_0}, \frac{\partial L}{\partial \Phi_1}\right]$$

Then we can update the parameters as follows:

$$\begin{cases} \Phi_0 \leftarrow \Phi_0 - \lambda \dfrac{\partial L}{\partial \Phi_0} \\ \Phi_1 \leftarrow \Phi_1 - \lambda \dfrac{\partial L}{\partial \Phi_1} \end{cases} \quad \Rightarrow \quad \Phi^{new} = \Phi^{old} - \lambda \nabla_{\{\Phi\}} L$$

where λ is the **learning rate**, a hyperparameter that controls the size of the parameter updates.

This is only a step in the optimization process. The optimization algorithm iteratively updates the parameters until the loss converges to a minimum. But when shell we stop?

$$|L^{new} - L^{old}| < \varepsilon$$

where ε is a small positive number that determines the convergence threshold.

Exercises

Let's talk further about the loss function:

$$L = \frac{1}{N}\sum_{i=1}^{N}(\underbrace{\Phi_0 + \Phi_1 x_i}_{=P_i} - y_i)^2$$

The aim is to minimize the loss function. In this case the loss is a paraboloid function of Φ_0 and Φ_1, so it has a single minimum.

Questions

1. Calculate the gradient of the Loss function
2. Find the minimum of the loss function
3. Show that the gradient of L is orthogonal to the level lines
4. Estimate the computational complexity of the exact solution of the linear regression problem in the general case (take into account the number of data samples and the number of dimensions/features used)

$$\Phi = (X^\top X)^{-1} X^\top y$$

Solutions

1. Calculate the gradient of the Loss function
2. Find the minimum of the loss function
3. Show that the gradient of L is orthogonal to the level lines
4. Estimate the computational complexity of the exact solution of the linear regression problem in the general case (take into account the number of data samples and the number of dimensions/features used)

   $$\Phi = (X^\top X)^{-1} X^\top y$$

   Let's consider a $X_{N\times D}$ matrix and let's computate the complexity of the single operations:
   - $(x^\top X)$ is a $D \times D$ matrix, so the complexity is $O(ND^2)$
   - $(x^\top X)^{-1}$ requires $O(D^3)$ operations
   - $(x^\top y)$ is a $D \times 1$ matrix, so the complexity is $O(ND)$

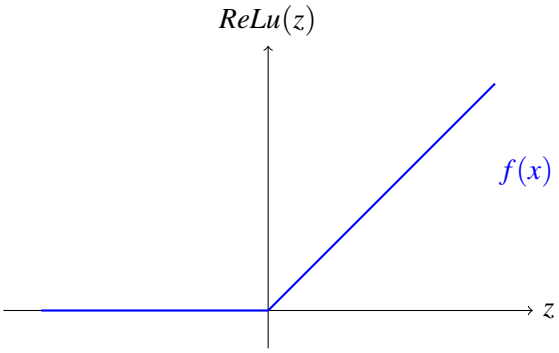
   The total complexity is $O(ND^2 + D^3 + \cancel{ND}) = O(ND^2 + D^3)$

2 Shallow Neural Networks

2.1 Activation Functions

2.1.1 Rectified Linear Unit (ReLU)

The ReLU activation function is defined as:



$$f(x) = \max(0,x) = \begin{cases} 0, & \text{if } x < 0 \\ x, & \text{if } x \geq 0 \end{cases}$$

Figure 2.1: Rectified Linear Unit (ReLU) activation function

Draft

2.1.2 Elements of the family F

$$y = \phi_0 + \phi_1 a[\theta_{10} + \theta_{11}x] + \phi_2 a[\theta_{20} + \theta_{21}x] + \phi_3 a[\theta_{30} + \theta_{31}x]$$

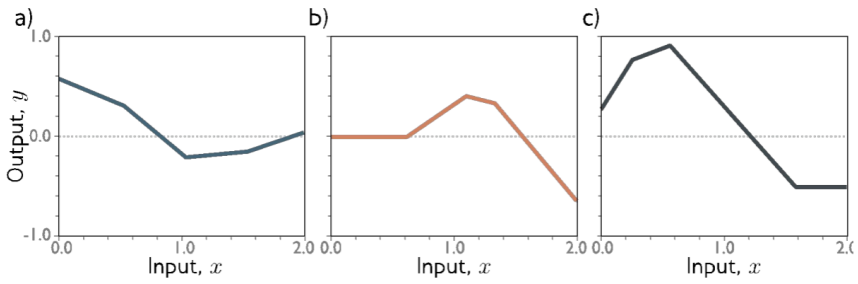
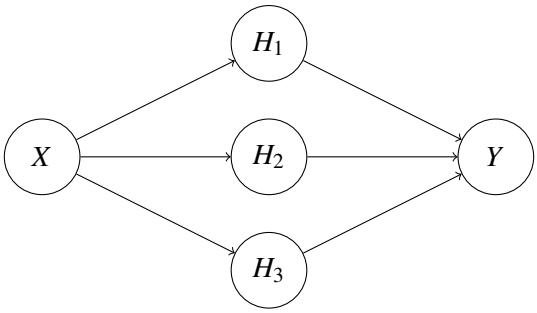


Figure 2.2: Family of functions F

$$\begin{cases} z_1 = \phi(a_1) = \phi_0 + \phi_1 a[\theta_{10} + \theta_{11}x] \\ z_2 = \phi(a_2) = \phi_0 + \phi_2 a[\theta_{20} + \theta_{21}x] \\ z_3 = \phi(a_3) = \phi_0 + \phi_3 a[\theta_{30} + \theta_{31}x] \end{cases}$$



$$y = W^{(2)}\phi(W^{(1)}x + b^{(1)}) + b^{(2)}$$

$$W^* = W^{(2)}W^{(1)} \qquad \text{rank}(W^*) = \min$$

…

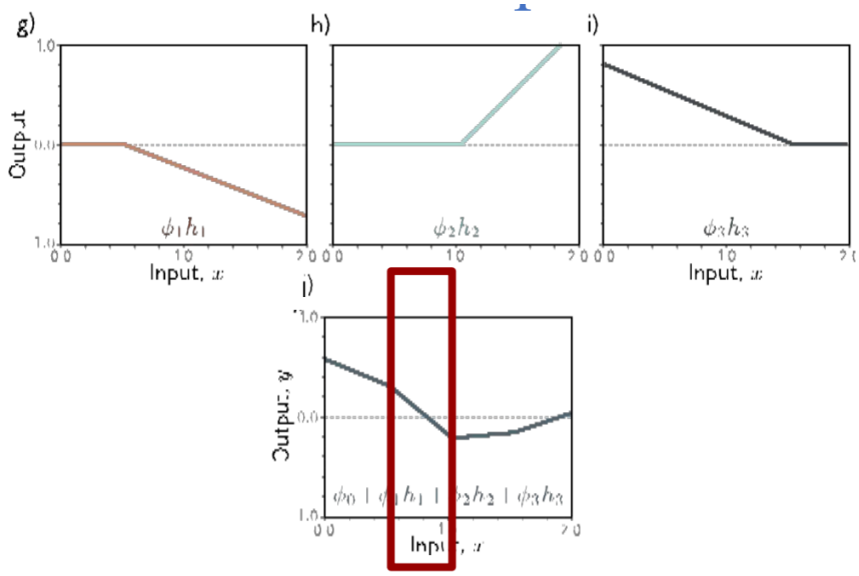
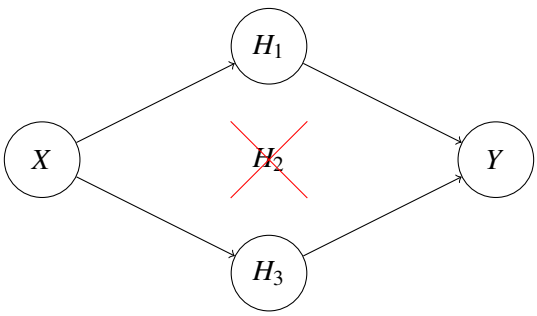


Figure 2.3: Flow computation

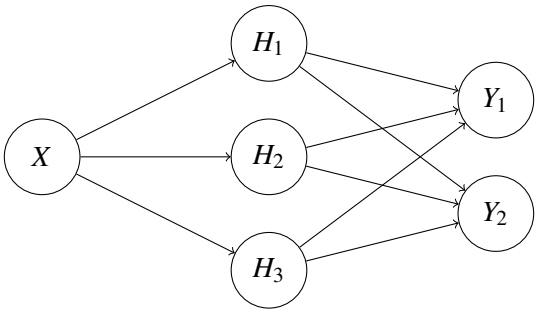
In the hilighted section, the contribute of the 2nd neuron of the hidden layer is zero, so the output is only influenced by the 1st and 3rd neurons of the hidden layer.

This is a consequence of the ReLU activation function, which deactivates the second node in that interval of the input space:



Multivariate Output

For multivariate output, the situation is similar to the previous case. In this case, the output layer has two neurons, Y_1 and Y_2:



We obtain the output as:

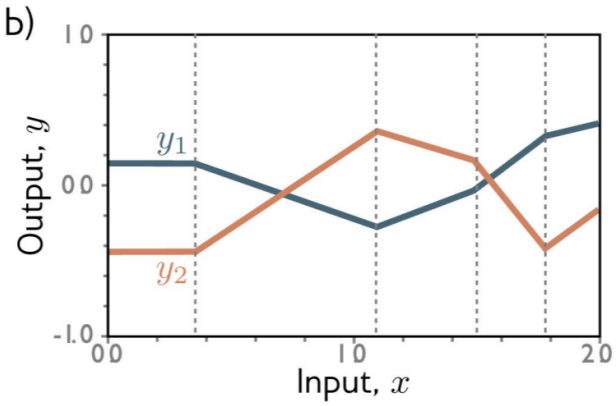
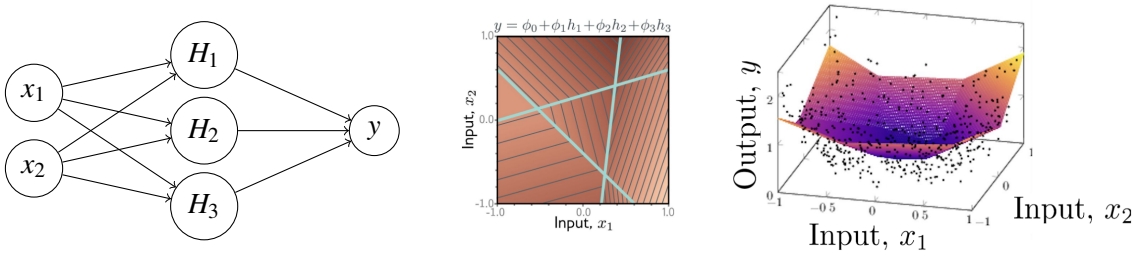


Figure 2.4: Flow computation for multivariate output

Multivariate Input

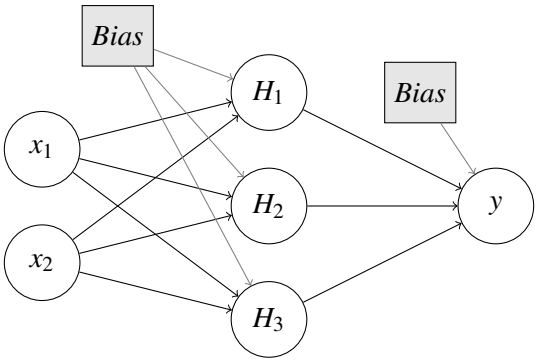


In this case we have 13 parameters:

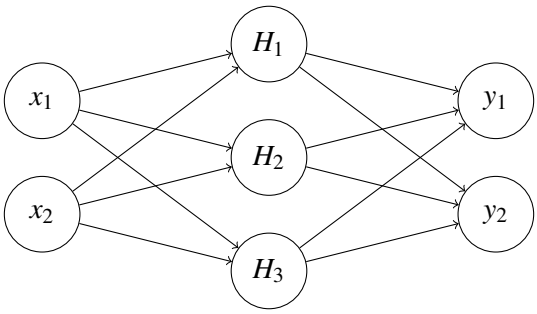
- We have 3 parameters from X_1 and X_2 to connect each node of the hidden layer (2 are the weights and 1 is the bias), for a total of 9 parameters.
- We have 4 parameters from the hidden layer to the output layer (3 weights and the bias term).

The general formula to count the number of parameters is:

#



General Case

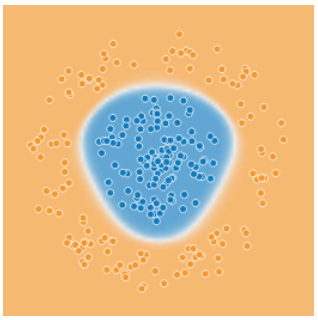


Draft

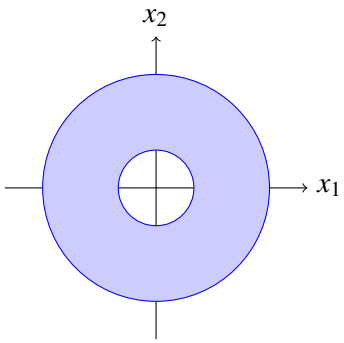
3 Lecture 18/03/2025

Polar Coordinates

Let's consider data disposed as in figure:

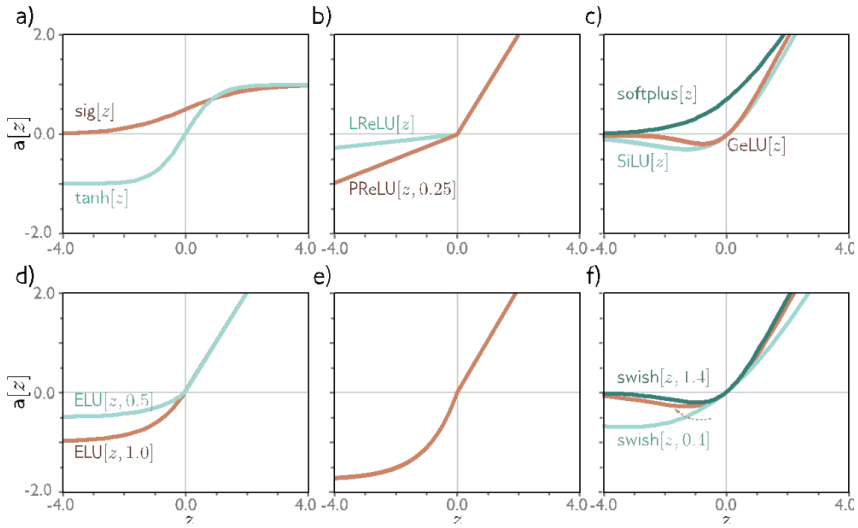


We can convert the data from Cartesian to Polar coordinates as follows:



$$\begin{cases} x_1 \\ x_2 \end{cases} \rightarrow \begin{cases} r = \sqrt{x_1^2 + x_2^2} \\ \theta = arctg\dfrac{x_2}{x_1} \end{cases}$$

Other Non-Linearities



3.1 Universal Approximation Theorem

Theorem 3.1. *Let σ be any continuous discriminatory function.*

$$\textit{Then finite sums of the form} \quad G(x) = \sum_{j=1}^{N} \alpha_j \sigma(y_i^\top x + \theta_j) \quad \textit{are dense in } C(I_n).$$

In other words, given any $f \in C(I_n)$ and $\varepsilon > 0$, there is a sum, $G(x)$, of the above form, for wich:

$$|G(x) - f(x)| < \varepsilon \qquad \forall x \in I_n$$

Bounds of the Universal Approximation Theorem

Any function $f(x)$ on $\mathbb{R}^d$ with some smoothness conditions can be approximated by a single hidden layer sigmoidal neural network $f_n(x)$ with n hidden units such that:

$$\int_{B_r} (f(x) - f_n(x))^2 \mu(dx) \leq \frac{c}{n}$$

where μ is a probability measure on ball $B_r = \{x : |x| < r\}$, c is a constant independent of n and r.

Observation: ***Feature Learning Advantage***

A result of the Universal Approximation Theorem is that no linear combinain of n fixed basis functions yells integrated square error smaller than order:

$$\left(\frac{1}{n}\right)^{\frac{2}{d}}$$

3.2 Number of Linear Regions

Be D_i the size of the input layer, D the size of the hidden layer, Then, given $D_i \leq D$.

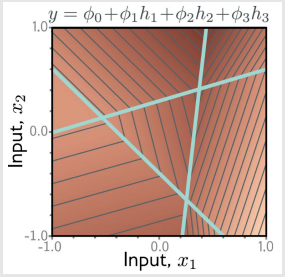
The number of linear regions N is given by:

$$N \leq \sum_{j=0}^{D_i} \binom{D}{j}$$

Example:

Let's consider $D_i = 2$ and $D = 3$. The number of linear regions is given by:

$$N \leq \binom{3}{0} + \binom{3}{1} + \binom{3}{2} = 1 + 3 + 3 = 7$$



Fixed Functions

$$y = \sum_{w_i} \phi(W_{i_{D_i}} x_{D_i} + b_i) \sim \{B_i(x)\}$$

Polynomial case

If the dimension is $DIM = 1$ we have something of the form:

$$y = w_0 + w_1 x + w_2 x^2 + \cdots + w_M x^M$$

If the dimension is $DIM = D$, we have:

$$y = w_o + \sum_{i=1}^{D} w_i x_i + \sum_{i,j=1}^{D} w_{ij} x_i x_j + \cdots + \sum_{i_1,\ldots,i_D=1}^{D} w_{i_1,\ldots,i_D} x_{i_1} \ldots x_{i_D}$$

The consequence is that the number of parameters grows exponentially with the dimension.

Curse of dimensionality

$$B_i(x) = \begin{cases} 0 & \text{if } x \text{ does not face in the block } b_i \\ majority & \text{if the class of } x \text{ in the block } A \end{cases}$$

Limitations of Fixed Basis Functions

High dimensionality introduces several challenges. One intuitive explanation comes from examining the probability density function in high-dimensional spaces.

Consider the marginal density of the radial coordinate:

$$p(r) = \int_{d\Omega} d\Omega \int p(x)\, dx,$$

where $d\Omega$ denotes the differential solid angle.

By switching to polar coordinates, we have:

$$p(x) \to p(r, \Omega),$$

so that the radial density becomes:

$$p(r) = \frac{S_D\, r^{D-1}}{(2\pi\sigma^2)^{D/2}} e^{-r^2/(2\sigma^2)},$$

with S_D representing the surface area of the unit sphere in D dimensions.

A key insight is that the probability mass concentrates around a specific radius. In fact, the mode of the radial density occurs approximately at:

$$r^* \approx \sigma\sqrt{D}.$$

Moreover, for a small deviation ε from r^*, the density decays as:

$$p(r^* + \varepsilon) = p(r^*) \exp\left(-\frac{3\varepsilon^2}{2\sigma^2}\right).$$

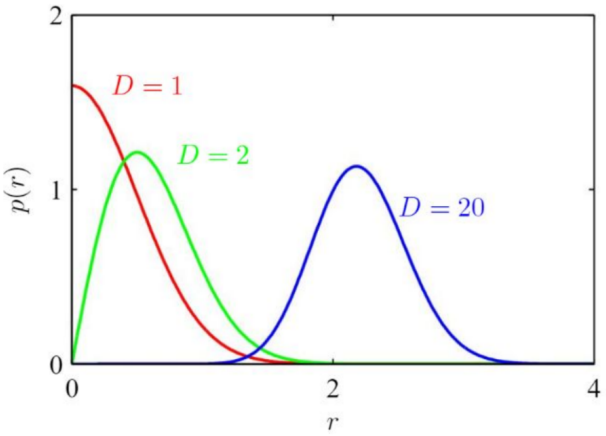


Figure 3.1: Curse of dimensionality.

Mainfolds hypothesis

Most ”naturally occuring” datasets lie on a low-dimensional mainfold

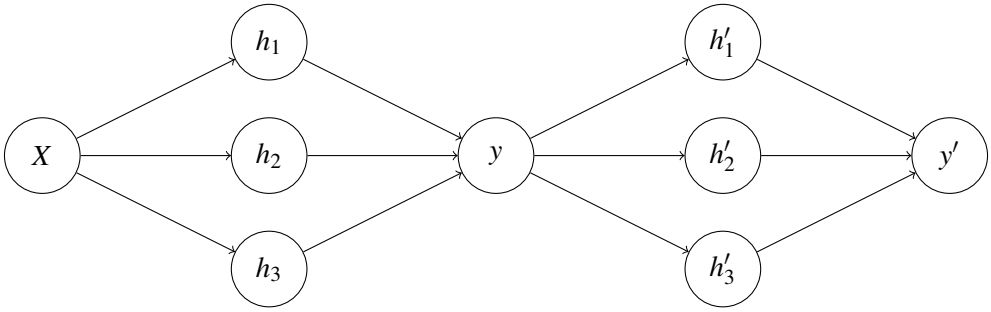
Draft

4 Deep Networks

4.1 Composition of shallow networks

The composition of shallow networks is a way to increase the number of linear regions. The idea is to stack multiple shallow networks to create a deep network.

Let's consider firstly a simple example with two shallow networks:



In this case we can distinguish how the two network behaves:

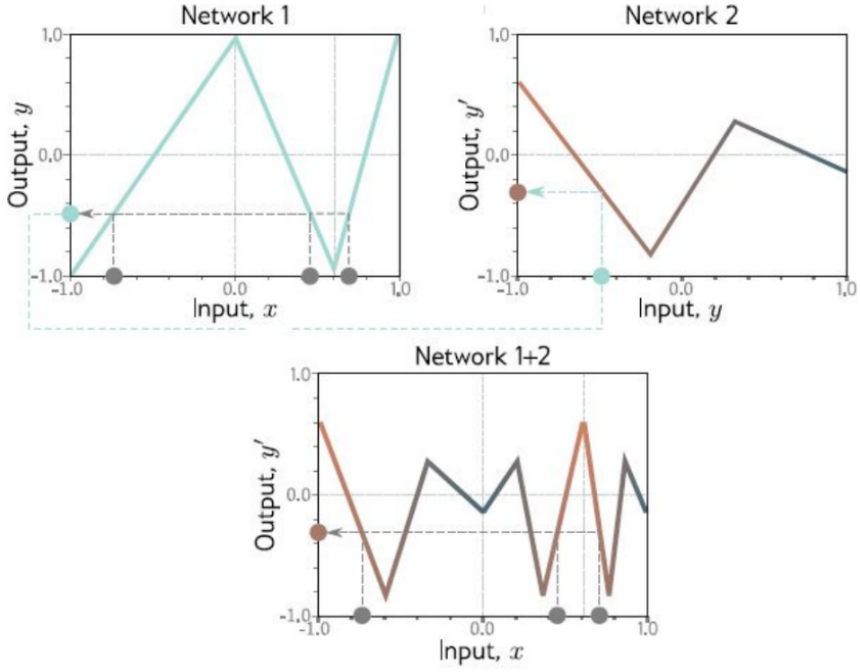
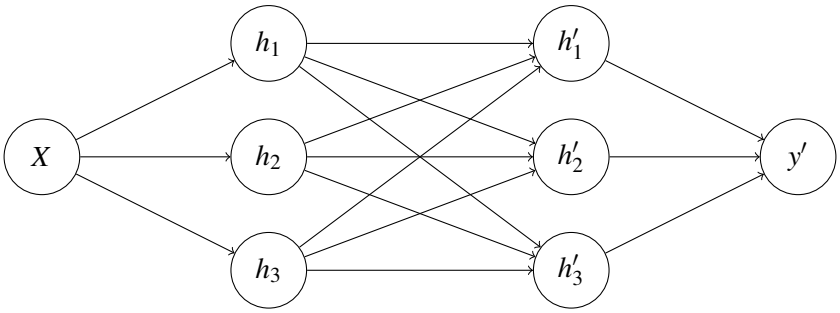


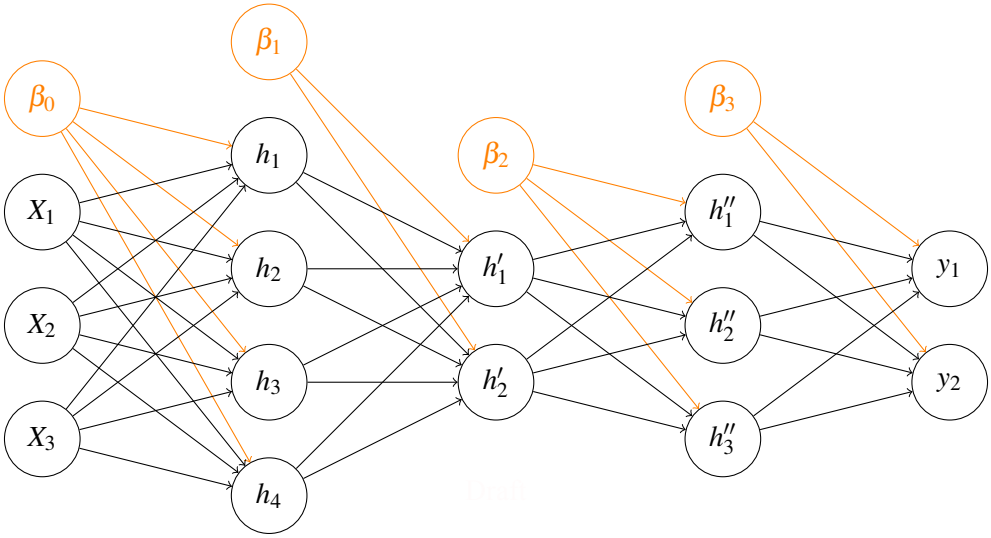
Figure 4.1: Composition of shallow networks.

We can reinterpret the composition of shallow networks as 2-layer a deep network:



4.2 Generic MLP

The generic Multi-Layer Perceptron (MLP) is a deep network with multiple hidden layers.



5 Loss Functions

Mean Squared Error (MSE)

Mean Squared Error is the most common loss function used for regression problems. MSE is the sum of squared distances between our target variable and predicted values.

$$L = \frac{1}{N}\sum_{i}^{N}\left(y_i - f(c^i;\{\phi\})\right)^2$$

Maximum Likelihood Estimation (MLE) is equivalent to minimizing the MSE loss function.

—

Recipe for loss construction:

1. Choose a suitable probability distribution $\Pr(y|\theta)$ that is defined over the domain of the predictions y and has distribution parameters θ
2. Set the mahchine learning model $f[x,\phi]$ to predict one or more of these parameters so $\theta = f[x,\phi]$ and $\Pr(y|\theta) = \Pr(y|f[x,\phi])$.
3. To train the model, find the network parameters $\hat{\phi}$ that minimize the negative log-likelihood over the training dataset pairs:

$$\hat{\phi} = \arg\min_{\phi}[L[\phi]] = \arg\min_{\phi}\left[-\sum_{i=1}^{N}\log\left[\Pr(y_i|f[x_i,\phi])\right]\right]$$

   This is equivalent to maximise the probability of the observed data under the model:

$$\Pr(\{x_i,y_i\}) = \prod_{i=1}^{N}\Pr(y_i|f[x_i;\{\phi\}])$$

...

Binary classification Problem

$$\begin{cases}\lambda = \Pr(y=1|x)\\ 1-\lambda = \Pr(y=0|x)\end{cases} \quad \Leftrightarrow \quad \Pr(y|x) = \lambda^y(1-\lambda)^{1-y}$$

$$\min_{\lambda} -\sum_{i}^{N} y^i\log\lambda + (1-y^i)\log(1-\lambda)$$

suppose that you want to classify a $\mathbb{R}^d$ dimentional data

$$\begin{matrix}x^{(1)},y^{(1)}\\ x^{(2)},y^{(2)}\\ \vdots\\ x^{(N)},y^{(N)}\end{matrix}$$

where $x^{(i)} \in \mathbb{R}^d$ and $y^{(i)} \in \{0,1\}$

In this case we cannot use a ReLu activation function in the output layer, because the output of the ReLu function is not bounded between 0 and 1. We need to use a ***sigmoid activation*** function in the output layer.

This is for a binary classification problem.

Multi-class classification Problem

For a multi-class classification problem we want to have multiple choices for the output layer.

Let's consider k parameters for the output layer. The values returned by the output layer are not probabilities, but scores, usually called ***logits***.

In this case we need to use a ***softmax activation*** function in the output layer.

$$\begin{array}{rcl} z_o & \rightarrow & e^{z_o} / \sum_i^k e^i \\ z_1 & \rightarrow & e^{z_1} / \sum_i^k e^i \\ & \vdots & \\ z_k & \rightarrow & e^{z_k} / \sum_i^k e^i \end{array}$$

The softmax function is a generalization of the sigmoid function and is used in the output layer of a neural network when we are dealing with a multi-class classification problem.

Information Theoretical Perspective on Loss Functions

We can calculate the empirical probability distribution as follows:

$$\Pr_{emp}(\{x\}) = \sum_{i=1}^{N} \delta(x - x_i)$$

where δ is the ***Dirac delta*** function.

…

We can calculate the "distance" between two probability distributions using the ***Kullback-Leibler divergence***:

$$KL[q][p] = \int_{-\infty}^{\infty} q(z) \log[q(z)] dz - \int_{-\infty}^{\infty} q(z) \log[p(z)] dz$$

This is not symmetric $KL[q][p] \neq KL[p][q]$ but it is always positive $KL[q][p] \geq 0$.

$$\min KL(q|p(\theta)) \qquad \Leftrightarrow \qquad MLE$$

0

$$KL(q|p(\theta)) = -\int \frac{1}{N} \sum_{i=1}^{N} \delta(x - x_i) \log p(x;\theta) dx + \underbrace{\cancel{\int \frac{1}{N} \sum_{i=1}^{N} \delta(x - x_i) \log \frac{1}{N} \sum_{i=1}^{N} \delta(x - x_i) dx}}_{\text{not dependent on } \theta}$$

So we can minimize the KL divergence by minimizing the negative log-likelihood.

—

We said that the KL divergence is not symmetric:

$$\min KL(p|q) = \min \sum_{i=1}^{N} p_i \log \frac{p_i}{q_i}$$

We will always have a distance between de empirical distribution and the model distribution:

$$KL(p|q) > 0 \quad = \text{"wrong code"}$$

This is becouse We have limited knowledge of the source, and this reflects in a non optimal compression of the data. The KL divergence is the difference between the optimal compression and the compression we are able to achieve.

5.1 Training

5.1.1 Stochastic Gradient Descent

The ***Stochastic Gradient Descent*** (SGD) is a method to minimize a function by iteratively moving in the direction of the negative gradient of the function at each point.

...

Gabor model

The Gabor model is a linear model that uses a set of Gabor filters to extract features from an image. The Gabor filters are a set of sinusoidal functions that are modulated by a Gaussian function.

$$f[x,\phi] = \sin\left[\phi_0 + \sum_{i=1}^{N} \phi_i x_i\right] e^{-\Sigma_{i=1}^{N} \phi_i^2 x_i^2}$$

Gradient descent gets to the global minimum if we start in the right "valley", otherwise it will get stuck in a local minimum or near a saddle point.

The idea is to add noise to the gradient to escape from the local minimum:

- Compute the gradient based on only a subset of the points: a ***mini-batch***.
- Work through datsset sampling without replacement.
- One pass through the data is called an ***epoch***.

In this way we calculate the gradient of the loss function for a mini-batch, which will likely be different from the gradient of the whole dataset. The hope is that the noise in the gradient will help us escape from the local minimum.

Property of SGD

- Can escape from local minima
- Adds noise, but still sensible updates as based on part of data
- Uses all data equally
- Less computationally expensive
- Seems to find better solutions

- Doesn't convere in traditional sense
- Learning rate schedule: learning rate decreases over time

Momentum

6 Convolutional Neural Networks

6.1 Introduction

Convolutional Neural Networks (CNNs) are specialized architectures built to handle the unique challenges of image data. Images are inherently large in dimensionality, requiring significant computation and memory, yet their neighboring pixels often share strong statistical relationships.

CNNs leverage this local connectivity through convolutional layers, where learnable filters slide over the image to detect meaningful patterns. By reusing filter parameters across the entire image, CNNs greatly reduce their overall complexity compared to fully connected networks. Furthermore, convolutional layers preserve important features even if the image shifts slightly, helping the network learn position-invariant representations.

This makes CNNs particularly effective for visual tasks like object recognition, segmentation, and more.

Image Kernels

Image kernels, also referred to as filters, are essential in the field of image processing. They enable the detection of edges, smoothing, and other transformations necessary for feature extraction in deep learning.

The key idea behind image kernels is to apply a small matrix (the kernel) to an image in a sliding window fashion. This process is known as ***convolution***. The kernel is a small matrix that contains weights, and it is applied to the image to produce a new image, often referred to as the ***feature map*** or ***output image***. The convolution operation involves multiplying the kernel values with the corresponding pixel values in the image and summing them up to produce a single output pixel.

Figure 6.1: Convolution operation with a kernel

Invariance and equivariance

Invariance in convolutional neural networks refers to the property that certain transformations in the input image (e.g., small translations) do not significantly affect the output. By using kernels and pooling, CNNs learn features that remain stable even if the object shifts slightly within the field of view.

Formally, a function $f[x]$ is *invariant* to a transformation $t[x]$ if:

$$f[t[x]] = f[x], \quad \forall x$$

In other words, the output of the function $f[x]$ is the same regardless of the transformation $t[x]$.

Equivariance indicates that a transformation applied to the input is reflected in the output but preserves the structure of the transformation. Convolution operations naturally ensure that shifting an input image leads to

a corresponding shift in the resulting feature map, allowing the network to detect features regardless of their position in the image.

Formally, a function $f[x]$ is *equivariant* to a transformation $t[x]$ if:

$$f[t[x]] = t[f[x]], \quad \forall x$$

This means that if we apply a transformation $t[x]$ to the input, the output will also be transformed in the same way.

6.2 Convolutional Networks for 1D inputs

Convolutional networks consist of a series of convolutional layers, each of which is equivariant to translation. They also typically include pooling mechanisms that induce partial invariance to translation.

Convolution is a mathematical operation that combines two functions to produce a third function. In the context of image processing, it involves applying a kernel (or filter) to an image to extract features or perform transformations.

1D Convolution operation

Convolutional layers are network layers based on the convolution operation. In 1D, a convolution transforms an input vector x into an output vector z so that each output z_i is a weighted sum of nearby inputs. The same weights are used at every position and are collectively called the ***convolution kernel or filter***. The size of the region over which inputs are combined is termed the kernel size. For a kernel size of three, we have:

$$z_i = w_1 x_{i-1} + w_2 x_i + w_3 x_{i+1}$$

where $[w_1, w_2, w_3]^\top$ is the kernel.

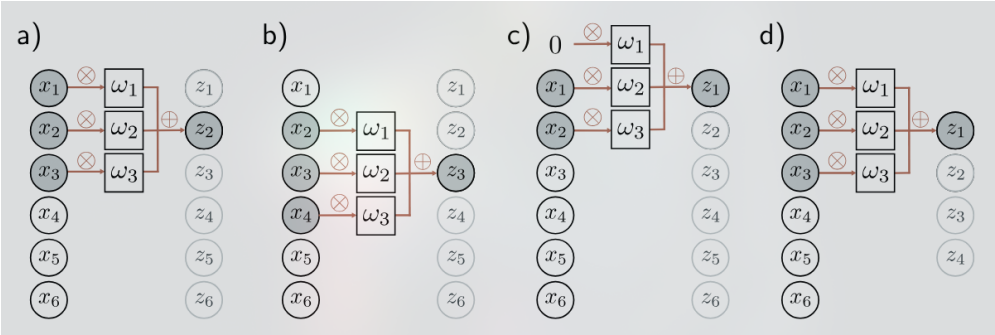


Figure 6.2: 1D convolution with kernel size three

More in general, we have $z_i = \sum_{j=0}^{k-1} w_j x_{i+j-k/2}$, where k is the kernel size, and w_j are the weights of the kernel.

Padding

As we have seen, each output is computed by taking a weighted sum of the previous, current, and subsequent positions in the input. This means that the first and last elements of the output are computed using fewer inputs. We can address this problem by adding **padding** to the input. The most common used is the ***Zero-padding***, which assumes the input is zero outsize its valid range.

Another approach is to discard the output positions where the kernel exceeds the range of input positions. These valid convolutions have the advantage of introducing no extra information at the edges of the input. However, they have the disadvantage that the representation decreases in size.

Stride, dilation, and kernel size

In the example above, each output was a sum of the nearest three inputs. However, this is just one of a larger family of convolution operations, the members of which are distinguished by their stride, kernel size, and dilation rate.

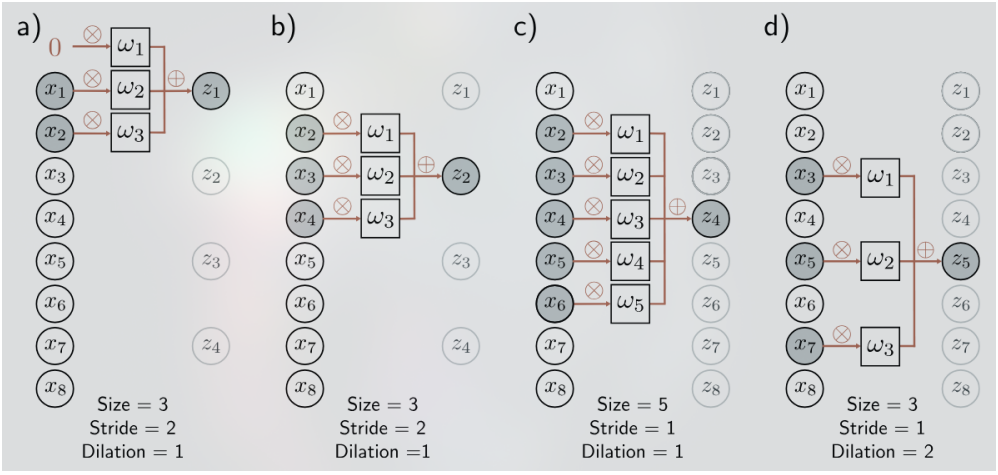


Figure 6.3: Stride, kernel size, and dilation

When we evaluate the output at every position, we term this a ***stride*** of one. However, it is also possible to shift the kernel by a stride greater than one. If we have a stride of two, we create roughly half the number of outputs

The ***kernel size*** can be increased to integrate over a larger area (Figure 6.3) However, it typically remains an odd number so that it can be centered around the current position. Increasing the kernel size has the disadvantage of requiring more weights. This leads to the idea of dilated or atrous convolutions, in which the kernel values are interspersed with zeros. For example, we can turn a kernel of size five into a dilated kernel of size three by setting the second and fourth elements to zero.

We still integrate information from a larger input region but onlu require three weights to do this. The number of zeros we intersperse between the weights determins the ***dilation rate***.

Convolutional layers

A convolutional layer computes its output by convolving the input, adding a bias , and passing each result through an activation function $a[\cdot]$. With kernel size three, stride one, and dilation rate one, the i^{th} hidden unit h_i would be computed as:

$$\begin{aligned} h_i &= a\left[\beta + w_1 x_{i-1} + w_2 x_i + w_3 x_{i+1}\right] \\ &= a\left[\beta + \textstyle\sum_{j=1}^{3} w_j x_{i+j-2}\right] \end{aligned}$$

Where the bias β and the weights w_j are learnable parameters.

This is a special case of a fully connected layer that computes the ith hidden unit as:

$$h_i = a\left[\beta_i + \sum_{j=1}^{D} w_{ij} x_j\right]$$

If there are D inputs and D hidden units, this requires D^2 weights. In contrast, the convolutional layer only requires three weights and one bias. A fully connected layer can reproduce this ecactly if most weights are set to zero and others are constrained to be identical.

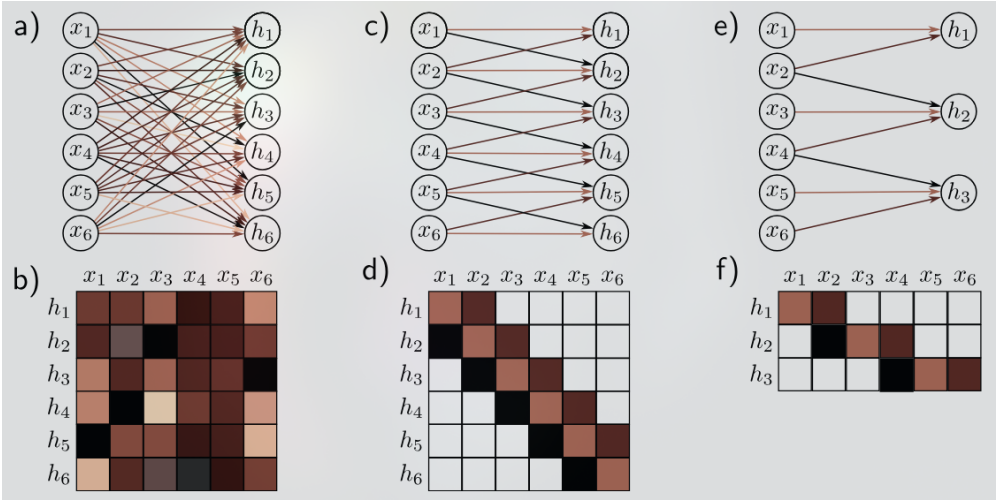


Figure 6.4: Fully connected vs. convolutional layers

Multiple channels

If we only apply a single convolution, information will likely be lost; we are averaging nearby inputs, and the ReLU activation function clips results that are less than zero. Hence, it is usual to compute several convolutions in parallel. Each convolution produces a new set of hidden variables, termed a ***feature map*** or ***channel***

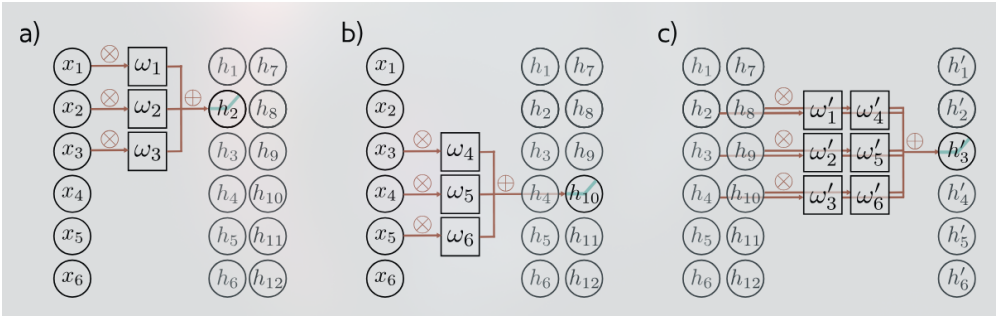


Figure 6.5: Multiple channels in a convolutional layer

Figure 6.5 illustrates this with two convolution kernels of size three and with zero-padding. The first kernel computes a weighted sum of the nearest three pixels, adds a bias, and passes the result through the activation function to produce hidde units h_1 to h_6. The second kernel computes a different weighted sum of the nearest three pixels, adds a different bias, and passes the results through the activation function to create hidden units h_7 to h_{12}.

In general, the input and the hidden layers all have multiple channels. If the incoming layer has C_i channels and we select a kernel size K per channel, the hidden units in each output channell are computed as a weighted sum over all C_i channels and K kernel entries using a weight matrix $\Omega \in \mathbb{R}^{C_i \times K}$ and one bias. Hence, if htere are C_o channels in the next layer, then we need $\Omega \in \mathbb{R}^{C_i \times C_o \times K}$ weights and $\beta \in \mathbb{R}^{C_o}$ biases.

Receptive fields

Convolutional networks comprise a sequence of convolutional layers. The ***receptive field*** of a hidden unit in the network is the region of the original input that feeds into it. Consider a convolutional network where each convolutional layer has kernel size three. The hidden units in the first layer take a weighted sum of the three closest inputs, so have receptive fields of size three. The units in the second layer take a weighted sum of the three closest positions in the first layer, which are themselves weighted sums of three inputs. Hence,

the hidden units in the second layer have a receptive field of size five. In this way, the receptive field of units in successive layers increases, and information from across the input is gradually integrated.

Draft

7 Lecture 15/04/2025

Problem Definition

Consider a network $f[x,\phi]$ with multivariate input x, parameters ϕ, and three hidden layers h_1, h_2, and h_3:

$$\begin{aligned} h_1 &= a[\beta_0 + \Omega_0 x] \\ h_2 &= a[\beta_1 + \Omega_1 h_1] \\ h_3 &= a[\beta_2 + \Omega_2 h_2] \\ f[x,\phi] &= \beta_3 + \Omega_3 h_3 \end{aligned}$$

where the function $a[\bullet]$ applies the activation function separately to every element of the input. The model parameters $\phi = \{\beta_0, \Omega_0, \beta_1, \Omega_1, \beta_2, \Omega_2, \beta_3, \Omega_3\}$ consist of the bias vectors β_k and the weight matrices Ω_k between every later.

We also have individual loss terms ℓ_i, which resultsrn the negative log-likelihood of the grount truth y_i given the model output $f[x_i,\phi]$. The total loss is the sum of the individual losses over the training set:

$$L[\phi] = \sum_i \ell_i$$

The most commonly used optimization algotirhm for training neural network is stochastic gradient descent (SGD), which updates the parameters as:

$$\phi_{t+1} \leftarrow \phi_t - a \sum_{i \in B_t} \frac{\partial \ell_i[\phi_t]}{\partial \phi}$$

where a is the learning rate, B_t contains the batch indices at iteration t. To compute this update, we need to compute the derivatives:

$$\frac{\partial \ell_i}{\partial \beta_k} \quad \text{and} \quad \frac{\partial \ell_i}{\partial \Omega_k}$$

for every parameters $\{\beta_k, \Omega_k\}$ at every layer $k \in \{0,1,...,K\}$ and for each index i in the batch B_t.

The backpropagation algorithm computes these derivatives efficiently by reusing the results of the forward pass. The algorithm is based on the chain rule of calculus, which allows us to compute the derivative of a function with respect to its parameters by breaking it down into smaller parts.

7.1 Computng Derivatives

The derivatives of the loss tell us how the loss changes when we make a small change to the parameters. Optimization algorithms exploit this information to manipulate the parameters so that the loss becomes smaller. The backpropagation algorithm computes these derivatives.

- **Observation 1.** Each weight (element of Ω_k) multiplies the activation at a source hidden unit and adds the result to a destination hidden unit in the next layer. It follows that the effect of any small change to the weight is amplified or attenuated by the activation at the source hidden unit. Hence, we run the network for each data example in the batch and store the activations of all the hidden units. This is known as the forward pass. The stored activations will subsequently be used to compute the gradients.

- **Observation 2.** A small change in a bias or weight causes a ripple effect of changes through the subsequent network. The change modifies the value of its destination hidden unit. This, in turn, changes the values of the hidden units in the subsequent layer, which will change the hidden units in the layer after that, and so on, until a change is made to the model output and, finally, the loss.

Hence, to know how changing a parameter modifies the loss, we also need to know how changes to every subsequent hidden layer will, in turn, modify their successor. These same quantities are required when considering other parameters in the same or earlier layers. It follows that we can calculate them once and reuse them.

7.1.1 Backpropagation algorithm

The backpropagation algorithm computes the derivatives of the loss with respect to the parameters in a neural network. The algorithm consists of two main steps: the forward pass and the backward pass.

Forward pass

In the forward pass, we compute the activations of all the hidden units and the model output for each data example in the batch. We store these activations for later use in the backward pass. The forward pass is done by applying the activation function to the weighted sum of the inputs at each layer.

$$\begin{aligned} f_0 &= \beta_0 + \Omega_0 x_i \\ h_1 &= a[f_0] \\ f_1 &= \beta_1 + \Omega_1 h_1 \\ h_2 &= a[f_1] \\ f_2 &= \beta_2 + \Omega_2 h_2 \\ h_3 &= a[f_2] \\ f_3 &= \beta_3 + \Omega_3 h_3 \\ \ell_i &= \ell[f_3, y_i] \end{aligned}$$

where f_{k-1} represents the pre-activations at the k^{th} hidden layer, and h_k contains the activations at the k^{th} hidden layer. The term $\ell[f_3, y_i]$ is the loss function.

In the forward pass, we work through these calculations and store all the intermediate quantities.

Backward pass

Now let's consider how the loss changes when the pre-activations f_0, f_1, f_2 change. Applying the chain rule, the expression for the derivative of the loss ℓ_i with respect to f_2 is:

$$\frac{\partial \ell_i}{\partial f_2} = \frac{\partial h_3}{\partial f_2}\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}$$

The three terms on the right-hand side have sized $D_3 \times D_3$, $D_3 \times D_f$, $D_f \times 1$, respectively, where D_3 is the number of hidden units in the last layer, D_f is the number of output units, and 1 is the size of the loss.

Similarly we can compute the loss changes when we change the pre-activations f_1 and f_0:

$$\begin{aligned} \frac{\partial \ell_i}{\partial f_1} &= \frac{\partial h_2}{\partial f_1}\frac{\partial f_2}{\partial h_2}\left(\frac{\partial h_3}{\partial f_2}\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right) \\ \frac{\partial \ell_i}{\partial f_0} &= \frac{\partial h_1}{\partial f_0}\frac{\partial f_1}{\partial h_1}\left(\frac{\partial h_2}{\partial f_1}\frac{\partial f_2}{\partial h_2}\frac{\partial h_3}{\partial f_2}\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right) \end{aligned}$$

Note that in each case, the term in brackets was computed in the previous step. By working backward through the network, we can reuse the previous computations.

Working backward through the right-hand side of the equations we have:

- The derivative $\partial \ell_i / \partial f_3$ of the loss ℓ_i with respect to the network output f_3 will depend on the loss function but usually has a simple form.
- The derivative $\partial f_3 / \partial h_3$ of the network output f_3 with respect to the last hidden unit h_3 is simply the weight matrix Ω_3:
$$\frac{\partial f_3}{\partial h_3} = \frac{\partial}{\partial h_3}(\beta_3 + \Omega_3 h_3) = \Omega_3^\top$$
- The derivative $\partial h_3 / \partial f_2$ of the last hidden unit h_3 with respect to the pre-activation f_2 depend on the activation function. It will be a diagonal matrix with the activation function's derivative on the diagonal. For ReLU functions, the diagonal terms are zero everywhere f_2 is less than zero and one otherwise. Rater than multiply by this matrix, we extract the diagonal terms as a vector $\mathbb{I}[f_2 > 0]$ and pointwise multiply, which is more efficient.
- The terms on the right-hand side of the last two equations have similar forms. As we progress back through the network, we alternately (i) multiply by the transpose of the weight matrices $\Omega_k^\top$ and (ii) threshold based on the input f_{k-1} to the hidden layer.

These inputs are stored during the forward pass.

Now that we know how to compute $\partial \ll_1 / \partial f_k$, we can focus on calculating the derivatives of the loss with respect to the weights and biases. To calculate the derivatives of the loss with respect to the biases β_k, we again use the chain rule:

$$\begin{aligned}\frac{\partial \ell_i}{\partial \beta_k} &= \frac{\partial f_k}{\partial \beta_k}\frac{\partial \ell_i}{\partial f_k} \\ &= \frac{\partial}{\partial \beta_i}(\beta_k + \Omega_k h_k)\frac{\partial \ell_i}{\partial f_k} \\ &= \frac{\partial \ell_i}{\partial f_k}\end{aligned}$$

which we already calculated.

Similarly, the derivative for the weights matrix Ω_k, is given by:

$$\begin{aligned}\frac{\partial \ell_i}{\partial \Omega_k} &= \frac{\partial f_k}{\partial \Omega_k}\frac{\partial \ell_i}{\partial f_k} \\ &= \frac{\partial}{\partial \Omega_i}(\beta_k + \Omega_k h_k)\frac{\partial \ell_i}{\partial f_k} h_k^\top \\ &= \frac{\partial \ell_i}{\partial f_k}\end{aligned}$$

Again, the progression from line two to line three is not obvious, however, the result makes sense.

The final line is a matrix of the same size as Ω_k. It depends linearly on h_k, which was multiplied by Ω_k in the original expression.

This is also consistent with the initial intuition that the derivative of the weights in Ω_k will be proportional to the values of the hidden units h_k that they multiply.

Summarizing

Consider a deep neural network $f[x_i, \phi]$ that takes input x_i, has K hidden layers with ReLU activations, and individual loss term $\ell_i = l[f[x_i, \phi], y_i]$. The goal of backpropagation is to compute the derivatives $\partial \ell_i / \partial \beta_k$ and $\partial \ell_i / \partial \Omega_k$ with respect to the biases β_k and weights Ω_k of the network.

Forward pass: We compute and store the following quantities:

$$\begin{aligned} f_0 &= \beta_0 + \Omega_0 \\ h_k &= a[f_{k-1}] & k \in \{0, 1, ..., K\} \\ f_k &= \beta_k + \Omega_k h_{k-1} & k \in \{0, 1, ..., K\} \end{aligned}$$

Backward pass: We start with the derivative $\partial \ell_i / \partial f_K$ of the loss function with respect to the output f_K and work backword through the network:

$$\begin{aligned} \frac{\partial \ell_i}{\partial \beta_k} &= \frac{\partial \ell_i}{\partial f_k} & k \in \{0, 1, ..., K\} \\ \frac{\partial \ell_i}{\partial \Omega_k} &= \frac{\partial \ell_i}{\partial f_k} h_k^\top & k \in \{0, 1, ..., K\} \\ \frac{\partial \ell_i}{\partial f_{k-1}} &= \mathbb{I}[g_{k-1} > 0] \odot \left(\Omega_k^\top \frac{\partial \ell_i}{\partial f_k} \right) & k \in \{0, 1, ..., K\} \end{aligned}$$

Draft

8 Lecture: 06/05/2025

8.1 Models for sequences

We are talking about models able to learn from sequences of data. For ”sequence” we mean a sequence of symbols, for instance:

- text, DNA, RNA, protein, etc.
- time series, etc.
- anything that can be represented as a sequence of symbols (e.g. an image)

Our models will need a ”vocabulary” of symbols. For instance, in the case of text, we can have a vocabulary of words or characters.

In these kinds of models, it is not suggested to use classical one hot encoding. The reason is that the vocabulary can be very large, and one hot encoding would create a very sparse representation of the data. Moreover, some symbols can be very similar to each other, and one hot encoding would not take this into account. For instance, in the case of text, the words ”cat” and ”dog” are very similar, but one hot encoding would represent them as completely different vectors.

We’ll use a different representation, called ”embedding”. An embedding is a dense representation of the data, where similar symbols are represented by similar vectors.

The embedding matrix is a matrix of size $|V| \times d$, where $|V|$ is the size of the vocabulary and d is the size of the embedding.

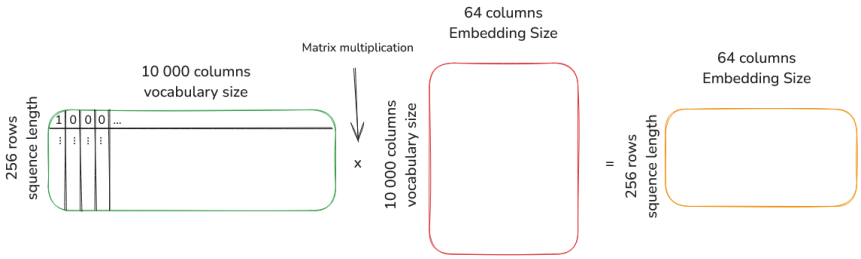


Figure 8.1: Embedding matrix

Let’s explore firstly the n-gram models. An n-gram model is a type of probabilistic language model used to predict the next item in a sequence based on the previous $n-1$ items. The model uses the conditional probability of the next item given the previous $n-1$ items, which can be represented mathematically as:

$$P(w_n|w_1, w_2, \ldots, w_{n-1}) = \frac{P(w_1, w_2, \ldots, w_n)}{P(w_1, w_2, \ldots, w_{n-1})}$$

where w_i represents the i-th word in the sequence.

For instance, a *bigram* model (where $n = 2$) would predict the next word based on the previous word, while a *trigram* model (where $n = 3$) would predict the next word based on the previous two words.

8.1.1 FF-Neural Language Models

A feedforward neural network language model (FF-NNLM) is a type of neural network used for language modeling. It uses a feedforward architecture to learn the probability distribution of sequences of words. The model takes a sequence of words as input and outputs a probability distribution over the next word in the sequence.

The FF-NNLM consists of an input layer (one hot encoded), an embedding layer, a hidden layer, and an output layer. The input layer represents the sequence of words as one-hot vectors, which are then passed through the embedding layer to obtain dense representations of the words. The hidden layer processes these representations and outputs a probability distribution over the next word in the sequence. The output layer uses a softmax activation function to produce the final probability distribution.

Recurrent neural networks

Recurrent neural networks (RNNs) are designed for processing sequences. They achieve this by incorporating a ”context vector” (or hidden state) that captures information from previous time steps. This context vector is updated at each step based on the current input and its previous state, and is fed back as an additional input to the network. This feedback mechanism allows RNNs to model long-term dependencies across the sequence.

When processing, for example, word embeddings, these are passed sequentially through a series of identical neural network units. Each unit produces an output (e.g., an output embedding) and an updated context vector. This updated context vector is then passed to the next unit in the sequence, along with the next input (e.g., the next word embedding), as depicted by the orange arrows in Figure 8.2.

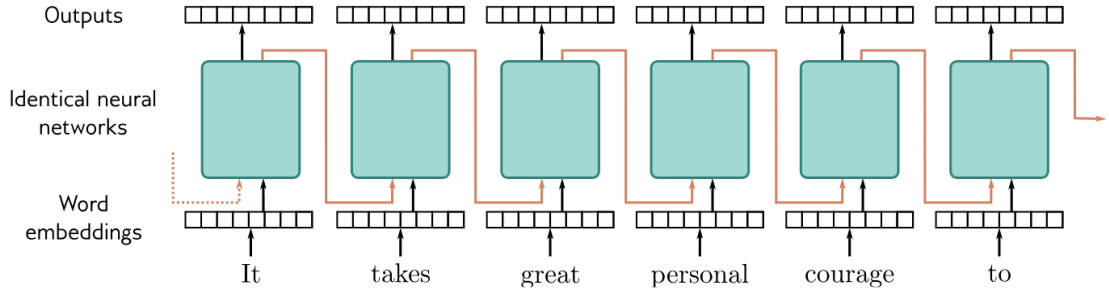


Figure 8.2: RNN

Each output embedding contains information about the word itself and its context in the preceding sentence fragment. In principle, the final output contains information about the entire sentence and could be used to support classification tasks similarly to the <cls> token in a transformer encoder model. However, RNNs sometimes gradually “forget” about tokens that are further back in time.

In principle, the ”context” vector could be used to represent the entire history of the sequence. However, in practice, RNNs have difficulty retaining information from earlier time steps, especially when the sequences are long. This is due to the vanishing gradient problem, where gradients become very small during backpropagation through time, making it difficult for the model to learn long-term dependencies.

The backpropagation algorithm for RNNs is called ***backpropagation through time*** (BPTT). It involves the calculation of the loss (e.g., cross-entropy loss) at each time step and then backpropagating the gradients through the entire sequence. This is done by unrolling the RNN for a fixed number of time steps and treating it as a feedforward network. The gradients are then calculated for each time step and accumulated to update the model parameters.

There are different types of RNNs, including *one to one*, *one to many*, *many to one*, and *many to many* architectures. The choice of architecture depends on the specific task and the nature of the input and output sequences.

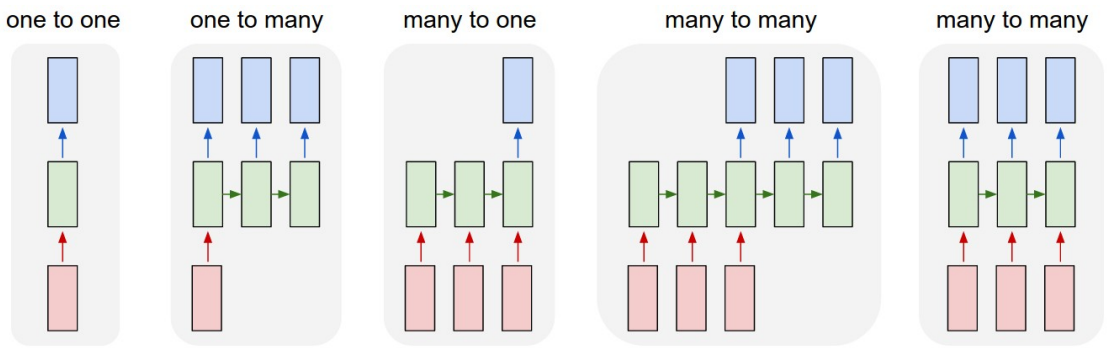


Figure 8.3: Types of RNNs

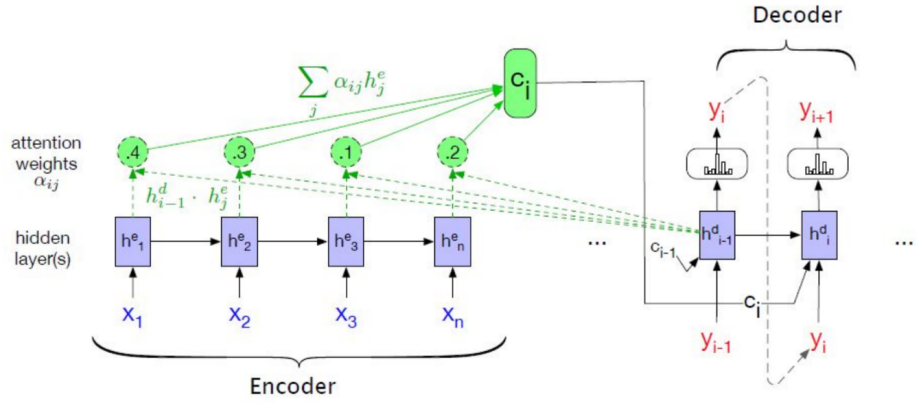
The case we were observing is the *many to one* case, where we have a sequence of inputs and a single output. This is typically used for tasks such as sentiment analysis, where we want to classify the entire sequence based on its content, or if we just want to predict the next word in a sequence.

Draft

9 Transformers

9.1 Lecture (13/05/2025)

Transformers are a type of neural network architecture that are particularly effective for natural language processing tasks. They are based on the *attention mechanism*, which allows the model to focus on different parts of the input sequence.



Let's consider an initial set of vectors $x_1, \ldots, x_n$, at each step of the attention mechanism, we want to compute a new set of vectors $y_1, \ldots, y_n$ (in the same space) which are a function of the x_i's. In this way, each new step creates another representation of the same input token.

Let's see now how to realize this kind of architecture.

A standard neural network layer $f[x]$, takes a $D \times 1$ input x and applies a linear transformation followed by an activation function like a ReLU, so:

$$f[x] = ReLU[\beta + \Omega x]$$

where β contains the biases, and Ω contains the weights.

A self-attention block $\mathbf{sa}[\bullet]$ takes N inputs $x_1, \ldots, x_N$, each of dimension $D \times 1$, and returns N outputs, each of which is also of size $D \times 1$. In the context of NLP, each input represents a word or word fragment. First, a set of values are computed for each input:

$$v_m = \beta_v + \Omega_v x_m$$

where $\beta_v \in \mathbb{R}^{D \times 1}$ and $\Omega_v \in \mathbb{R}^{D \times D}$ represent biases and weights, respectively.

Then the n^{th} output $\mathbf{sa}_n[x_1, \ldots, x_N]$ is a weighted sum of all the values $v_1, \ldots, v_N$:

$$\mathbf{sa}_n[x_1, \ldots, x_N] = \sum_{m=1}^{N} a[x_m, x_n] v_m$$

The scalar weight $a[x_m, x_n]$ is the attention that the n^{th} output pays to input x_m. The N weights $a[\bullet, x_n]$ are non-negative and sum to one. Hence, self-attention can be thought of as routing the values in different proportions to create each output.

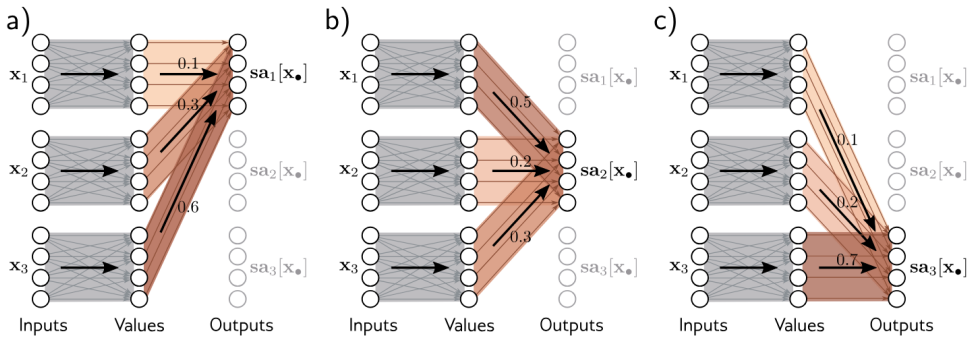


Figure 9.1: The self-attention mechanism takes N inputs $x_1,\ldots,x_N \in \mathbb{R}^D$ (here $N=3$ and $D=4$) and processes each separately to compute N value vectors. The n^{th} output $\mathbf{sa}_n[x_1,\ldots,x_N]$ ($\mathbf{sa}_n[x_\bullet]$ for short) is then computed as a weighted sum of the N value vectors, where the weights are positive and sum to one. [1]

The value vectors $\beta_v + \Omega_v x_m$ are computed independently for each input x_m, and these vectors are combined linearly by the attention weights $a[x_m, x_n]$. However, the overall self-attention computation is nonlinear.

This is an example of a ***hypernetwork***, where one network branch computes the weights of another.

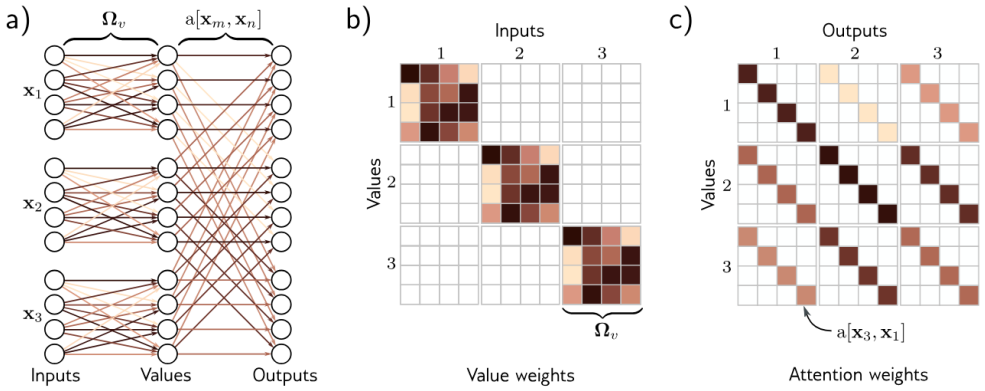


Figure 9.2: Self-attention for N = 3 inputs x_n, each with dimension $D = 4$.

To compute the attention, we apply two more linear transformations to the inputs:

$$q_n = \beta_q + \Omega_q x_n, \qquad k_m = \beta_k + \Omega_k x_m$$

where $\{q_n\}$ and $\{k_m\}$ are termed queries and keys, respectively. Then we compute dot products between the queries and keys and pass the results through a softmax function:

$$a[x_m, x_n] = \text{softmax}_m[k_\bullet^\top q_n] = \frac{\exp(k_\bullet^\top q_n)}{\sum_{m'} \exp(k_{m'}^\top q_n)}$$

so for each x_n, they are positive and sum to one. This is known as ***dot-product self-attention***.

The dot product operation returns a measure of similarity between its inputs, so the weights $a[x_m, x_n]$ depend on the relative similarities between the n^{th} query and all of the keys. The softmax function means that the key vectors "compete" with one another to contribute to the final result.

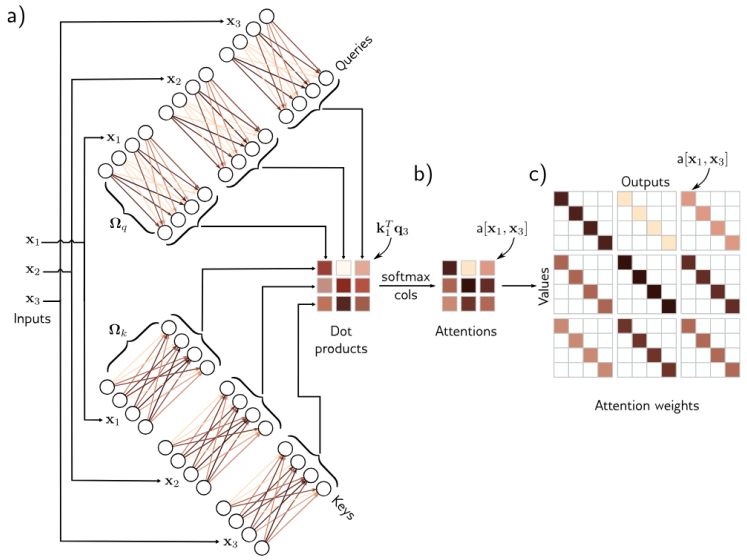


Figure 9.3: Computing attention weights: (a) Compute the query, key, and value vectors for each input. (b) Apply softmax to query-key dot products to form non-negative attentions that sum to one. (c) These route the value vectors via the sparse matrix from Figure 9.2.

9.2 The transformer layer (Lecture 20/05/2025)

The self-attention mechanism is only a part of a larger ***transformer layer***. This consist of a multi-head self-attention unit, followed by a fully connected network $mlp[x_\bullet]$; Both units are residual networks. In addition, it is typical to add a LayerNorm operation after both the self-attention and fully connected networks.

The LayerNorm is similar to the batch norm, but normalized each embedding in each batch element separately, using statictics calculated across its D dimensions.

Since we have:

$$f_k = \beta_k + \Omega_k a(f_{k-1})$$

this means that the norm of the k^{th} layer is much lower than the norm of the $(k-1)^{th}$ layer:

$$\|f_k\| < \|f_{k-1}\| \qquad \text{or} \qquad \|f_k\| < \frac{1}{2}\|a(f_{k-1})\|$$

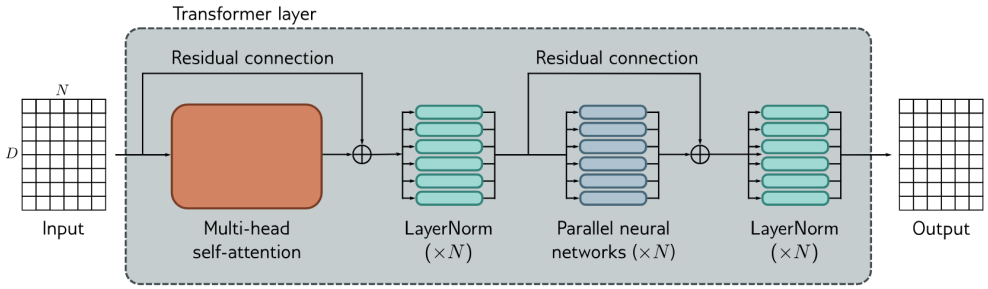


Figure 9.4: A transformer layer.

…

$$\mathbf{E}(f'_j) = \mathbf{E}(\beta_j + \sum_e \Omega_{je} h_e) = \underbrace{\mathbf{E}(\beta_j)}_{=0} + \mathbf{E}(\sum_e \underbrace{\Omega_{je} h_e}_{=0}) = 0$$

$$\sigma^2(f'_j) = \mathbf{E}(f'^2_j) - \cancel{(\mathbf{E}(f'_j))^2} = \sum_e \sum_j \mathbf{E}(\Omega_{ij}\Omega_{ie})\mathbf{E}(h_j h_e)$$

…

$$\sigma^2_{f'} = \frac{1}{2} D_h \cdot \sigma^2_\Omega \cdot \sigma^2_f \qquad \Rightarrow \qquad 1 = \frac{1}{2} D_h \cdot \sigma^2_\Omega \qquad \Rightarrow \qquad \boxed{\sigma^2_\Omega = \frac{2}{D_h}}$$

Transformer layers consist of two residual networks stacked together. Without proper normalization, the residual connections would experience exponential growth in size as they propagate through the network. To address this issue, we apply BatchNorm before each residual connection, which constrains the growth to be linear rather than exponential.

In this way we have that, at initialization, the deeper nodes of the network contribute less to the output, creating a sort of ”effective depth” in the first layer. As the training progresses, the deeper nodes contribute more to the output, and the effective depth increases.

9.3 Transformers for NLP

A typical NLP pipeline starts with a tokenizer, which are mapped to a learning embedding, and then pass through different transformer layers.

There are different architectures we are interested in:

- Encoder (BERT)
- Decoder (GPT)
- Encoder-Decoder
- IMG

9.3.1 Encoder model: BERT

BERT is an encoder model which uses a vocabolary of about 30.000 tokens. Input tokens are converted into 1024 dimensional word embeddings and passed through 24 transformer layers. Each layer contains a mechanism of 16 heads. The total amount of parameters is about 340 million.

The input tokens (and a special <cls> token denoting the start of the sequence) are converted to word embeddings. These embeddings are passed through a series of transformer layers to create a set of output embeddings.

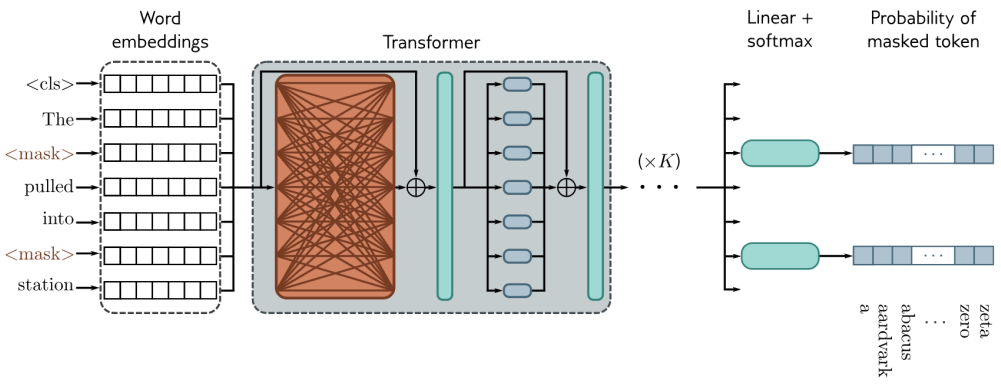


Figure 9.5: Pre-training for BERT-like encoders.

Pre-training

In the pre-training stage, the network is trained using self-supervision. For BERT, the self-supervision task consists of predicting missing words from sentences from a large internet corpus, forcing the transformer

network to understand some syntax. A small fraction of the input tokens are randomly replaced with a generic <mask> token. In pre-training, the goal is to predict the missing word from the associated output embedding. To this end, the outputs corresponding to the masked tokens are passed through softmax functions, and a multiclass classification loss is applied to each. This task has the advantage that it uses both the left and right context to predict the missing word but has the disadvantage that it does not make efficient use of data; here, seven tokens need to be processed to add two terms to the loss function.

After pre-training, the encoder is fine-tuned using manually labeled data to solve a particular task. Usually, a linear transformation or a multi-layer perceptron (MLP) is appended to the encoder to produce whatever output is required.

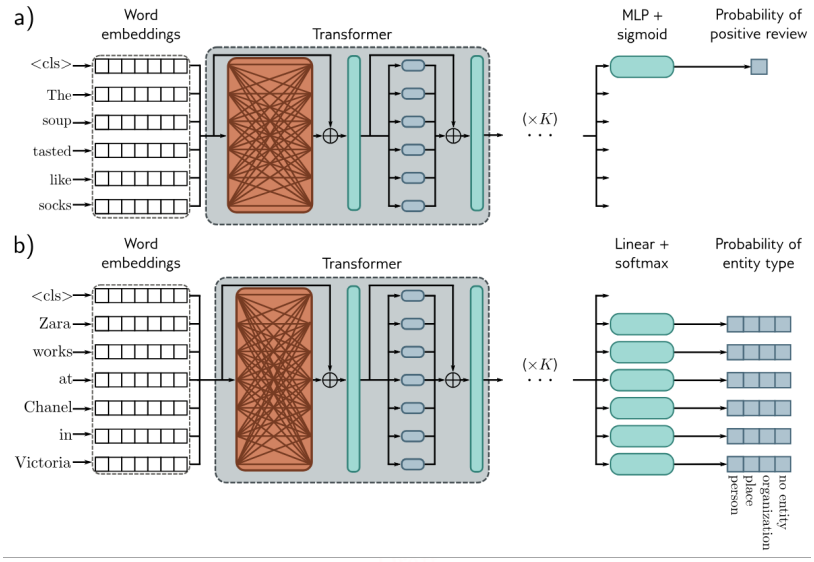


Figure 9.6: Pre-training for BERT-like encoders.

9.3.2 Decoder model: GPT

The decoder is a transformer model with a single attention head. It's aimed to predict the next word of a sentence, and generate a coherent text passage by feeding the extended sequence back into the model.

An autoregressive model predicts the conditional distributions $\Pr(t_n|t_1,\ldots,t_{n-1})$ for each token given all the prior ones, and hence, indirectly, computes the joint distribution $\Pr(t_1,\ldots,t_N)$ of all the N tokens:

$$\Pr(t_1,\ldots,t_N) = \Pr(t_1)\prod_{n=2}^{N}\Pr(t_n|t_1,\ldots,t_{n-1})$$

The autoregressive formulation demonstrates the connection between maximizing the joint probability of the tokens and the next token prediction task.

Masked self-attention

To train a decoder, we seek parameters that maximize the log probability of the input text under the autoregressive model. Ideally, we would like to replace all the log probabilities and gradients in the same forward pass rather than doing a forward pass for each token in the sentence. However, if we pass in the full sentence, the term computing $\log[\Pr(t_{next}|\textit{all tokens})]$ would have access to both the answer and the right context, so the model would simply copy the word after the one we want to learn, and won't train properly.

Fortunately, the tokens only interact in the self-attention layers in a transformer network. Hence, the problem can be resolved by setting the corresponding self-products in the self-attention computation to zero so that the attention is only allowed between tokens before the answer. The input tokens are passed through the softmax function. This is known as **masked self-attention**.

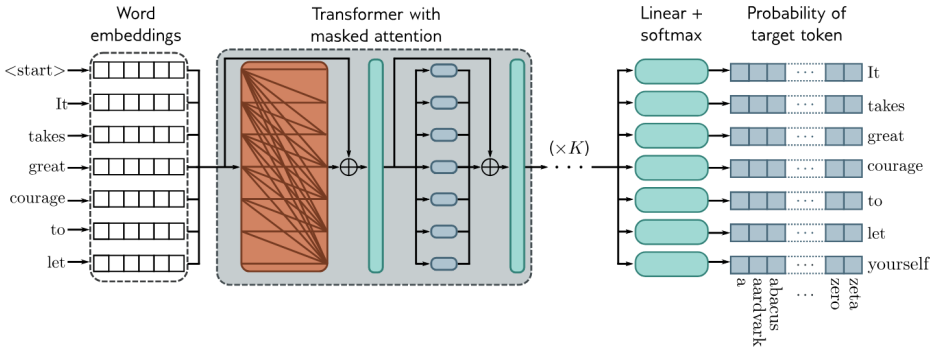


Figure 9.7: Training GPT-type decoder with masked self-attention. Each token can only attend to previous tokens (orange connections) to predict the next token in sequence.

The decoder network operates through the following process:

- Input text is first tokenized and converted into ***embeddings***.
- These embeddings are processed through transformer layers that employ ***masked self-attention***, ensuring that each token can only attend to itself and previous tokens in the sequence.
- Each output embedding represents the context up to that position in the sequence, with the objective of ***predicting the subsequent token***.
- Following the transformer layers, a linear projection maps each output embedding to vocabulary size, and a softmax function transforms these logits into probability distributions over the ***vocabulary***.

During training, the objective is to maximize the log-likelihood of the correct next tokens at each position in the ground truth sequence, accomplished through standard cross-entropy loss optimization.

Few-Shot Learning

Since the autoregressive language model defines a probability model over text sequences, it can be used to sample new examples of plausible text. To generate from the model, we start with an input sequence of text (which might be just the special <start> token indicating the beginning of the sequence) and feed this into the network, which then outputs the probabilities over possible subsequent tokens. The new extended sequence can be fed back into the decoder network to yield the probability distribution over the next token. By repeating this process, we can generate large bodies of text.

Large language models such as GPT-3 implement these concepts at an unprecedented scale. A characteristic of models trained at this scale is their ability to execute various tasks without requiring fine-tuning. When presented with multiple examples of accurate question-answer pairs followed by a new question, these models frequently provide correct responses to the final question by extending the sequence pattern.

Example: ***Correcting the grammar***

Train:

- **Poor English input:** I eated the purple berries.
- **Good English output:** I ate the purple berries.
- **Poor English input:** Thank you for picking me as your designer. I'd appreciate it.
- **Good English output:** Thank you for choosing me as your designer. I appreciate it.
- **Poor English input:** The mentioned changes have done. or I did the alteration that you requested. or I changed things you wanted and did the modifications.
- **Good English output:** The requested changes have been made. or I made the alteration that you requested. or I changed things you wanted and made the modifications.

Test:

- **Poor English input:** I'd be more than happy to work with you in another project.
- **Good English output:** I'd be more than happy to work with you on another project.

Encoder-Decoder Architecture

Translation between languages is an example of a *sequence-to-sequence* task. One common approach uses both an encoder (to compute a good representation of the source sentence) and a decoder (to generate the sentence in the target language). This is aptly called an ***encoder-decoder model***.

The encoder-decoder architecture operates through the following process:

- The ***encoder*** receives the source sentence and processes it through transformer layers to create output representations for each token.
- The ***decoder*** receives the target sequence and processes it through modified transformer layers that employ masked self-attention to predict the next token at each position.
- Crucially, decoder layers also attend to encoder outputs through ***cross-attention***, where a new attention layer is inserted between the masked self-attention and feed-forward components.
- In cross-attention, queries are computed from decoder embeddings while keys and values derive from encoder embeddings, enabling the decoder to access source sequence information.

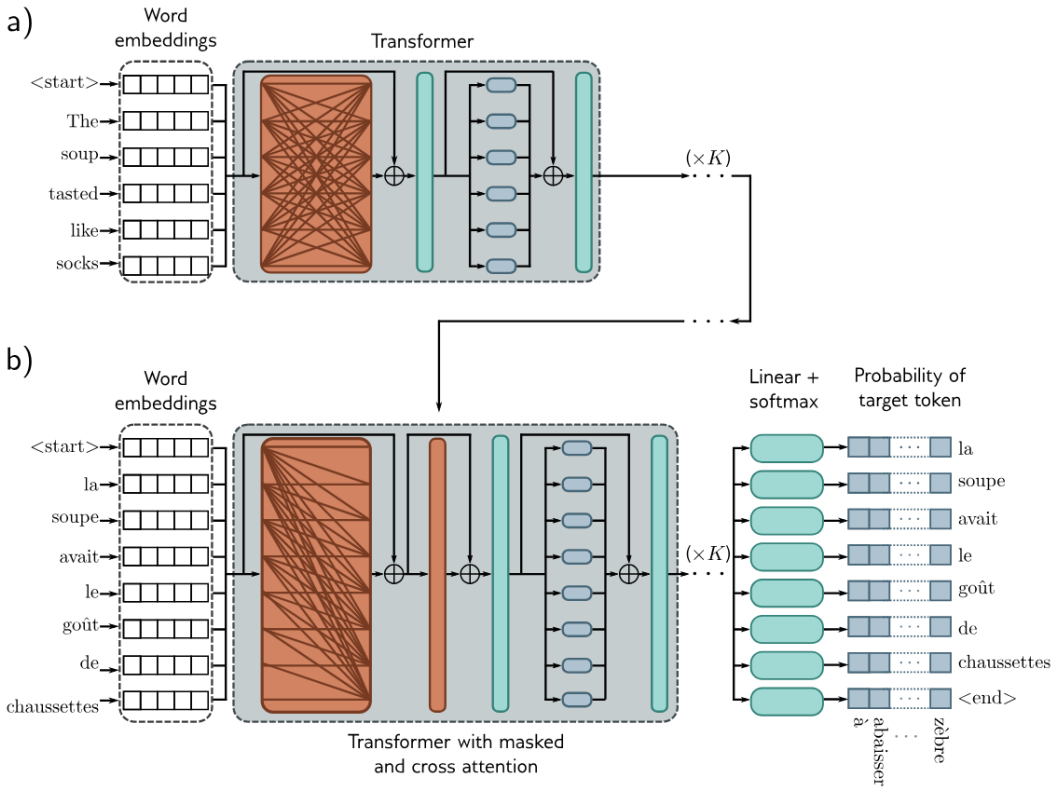


Figure 9.8: Encoder-decoder architecture. Two sentences are passed to the system with the goal of translating the first into the second. a) The first sentence is passed through a standard encoder. b) The second sentence is passed through a decoder that uses masked self-attention but also attends to the output embeddings of the encoder using cross-attention (orange rectangle). The loss function is the same as for the decoder model; we want to maximize the probability of the next word in the output sequence.

Transformers fot long sequences

…

9.3.3 Transformers for images

…

Vision Transformers (ViT)

The Vision Transformer tackled the problem of image resolution by dividing the image into 16×16 patches (Figure 9.9). Each patch is mapped to an input embedding via a learned linear transformation, and these representations are fed into the transformer network

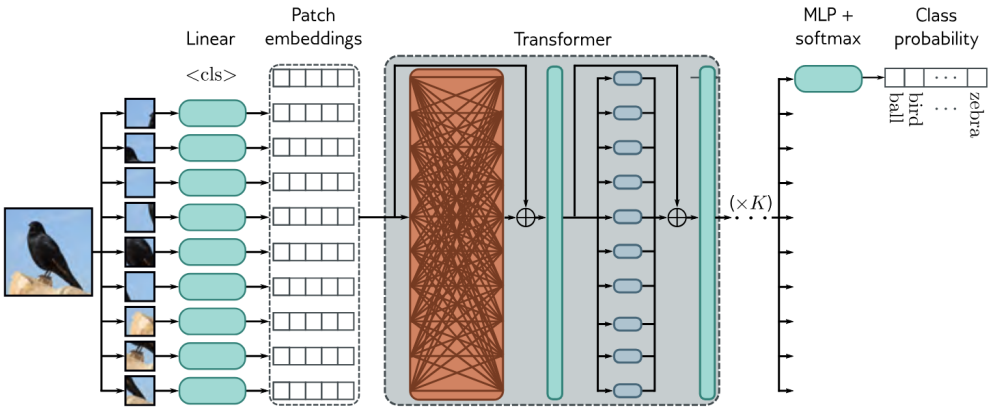


Figure 9.9: Vision Transformer architecture.

…

Bibliography

[1] Simon J.D. Prince. *Understanding Deep Learning*. The MIT Press, 2023. URL: http://udlbook.com.

Draft