CONTENTS

Part I

THESIS

1 INTRODUCTION TO NEURAL NETWORKS

This chapter introduces artificial neural networks, systems inspired by their biological counterparts which constitute animal brains. At the core of these biological systems are neurons: cells capable of transmitting electrochemical signals through a network. The architecture of artificial neural networks abstracts and simplifies these biological processes [Agg23]. The chapter begins by establishing a foundational understanding of artificial neural networks within the context of computational graphs. This perspective is especially advantageous when examining the backpropagation algorithm for neural network training in the last Section 1.3. Section 1.2 re-examines the components of neural networks discussed at the start of this chapter to illustrate them with examples and to re-state them in a precise mathematical way. This iterative and gradual approach was chosen to give the reader a gentle introduction to this complex topic. By the conclusion of this chapter, the reader will have acquired both a rigorous mathematical but also an intuitive and visual understanding of multi-layer neural networks, a prerequisite for the theoretical investigation of large language models in Chapter 2 and the exploration of algorithmic advances in Chapter 3.

1.1 UNDERSTANDING NEURAL NETWORKS

Neural networks often seem complex at first glance. To demystify them, this chapter starts with the concept of computational graphs. Afterward, we discuss single-layer neural networks as a stepping stone to the more complex multi-layer networks. As we progress, the terminology evolves, starting with basic terms and simplified notation, and step by step we introduce more technical language and refined notation. Over the course of this section, the computational graph abstraction of neural networks established in Subsection 1.1.1 will be gradually re-visited and re-interpreted in order to culminate in a rigorous mathematical description of multi-layer neural networks (Subsection 1.1.4).

1.1.1 *Computational Graphs*

Artificial neural networks are a computational model designed to execute through *directed acyclic computational graphs*. To set the stage for these graphs we start with Definition 1.

Definition 1 (Topological Sort) A *topological sort* is an arrangement of the vertices in a *directed acyclic graph* where each vertex represents an individual computational step such that for every directed edge $(u, v) \in E$ from vertex

$u \in U$ to vertex $v \in V$, u comes before v in the ordering. This ordering ensures that each computation represented by a vertex u precedes any subsequent steps represented by a vertex v. This is only possible if the graph has no circular dependencies.

In other words, a topological sort is a graph traversal in which each node v is visited only after all its dependencies are visited. Next, we present the foundational concept that - as we will see later - organizes the operations of a neural network:

Definition 2 (Directed Acyclic Computational Graph) A *directed acyclic computational graph*, denoted G, is a graph that represents a sequence of operations. It consists of a set of vertices V and a set of directed edges E. Each edge in E is directed from one vertex to another, such that there are no cycles, and a topological sort exists. These vertices represent computational operations and variables, and the edges represent the flow of data or dependencies between these operations. In computational graphs, the edges carry *weights*, denoted $w \in \mathbb{R}$, which are parameters that can be learned and adjusted.

In a directed acyclic computational graph, the computation begins when *input vertices* transmit their fixed input values to adjacent nodes through directed edges. The weights associated with the edges scale these values and are critical to determine the strength and sign of the influence one node has over another. Unlike input nodes, which simply serve as data entry points, the subsequent *computational nodes* in the graph execute mathematical operations on their inputs and pass their results to their neighboring vertices. The final output of these computations is collected at *output nodes*. Thus, a computational graph computes a vector to vector function $f : \mathbb{R}^{in} \to \mathbb{R}^{out}$ where in corresponds to the number of input nodes and out to the number of output nodes. Note that in order to compute f the weights must have some value from the beginning. If we do not have prior knowledge, usually, they are initialized from a random distribution (see Section 3.2 for details).

An example of a computational graph with three input nodes, two computational nodes and two output nodes is shown in Figure 1.1. Edges between input nodes and computational nodes are associated with weights w_1, w_2, w_3. The weights on the edges between the computational nodes and the output nodes are set to 1 and not displayed. A possible topological ordering is: $(x_1, x_2, x_3, mul_1, add_1, mul_2, add_2)$. For ease of representation, the set of values at the input nodes is commonly represented as an input vector $\vec{x} \in \mathbb{R}^{in}$, while the collection of outputs is similarly encapsulated in an output vector $\vec{o} \in \mathbb{R}^{out}$.

In a computational graph, nodes often perform calculations that are more complex than simple arithmetic operations. Formally, let us consider in to be the number of these input signals, where each signal x_i is a numerical value provided to the node. Then, the node multiplies each input signal x_i by a the scalar w_i, where the term w_i represents the weight assigned to the edge that connects the i-th input variable x_i to the computational node. These weighted inputs are then summed, and to this sum, the node adds a

constant term known as a *bias* (b), which allows the node to adjust the output independently of its inputs. Mathematically, the computation done by the node before applying any further transformations is called *pre-activation* (h_{pre}), calculated as:

$$h_{pre} = \sum_{i=1}^{in} x_i w_i + b,$$

After computing the pre-activation, the node applies an activation function $\Phi : \mathbb{R} \to \mathbb{R}$, resulting in what is known as the *post-activation* (h_{post}):

$$h_{post} = \Phi(h_{pre}) \, .$$

This post-activation value h_{post} is the final output of the node, which then serves as input to subsequent nodes in the graph. The activation function Φ introduces nonlinearity into the system, allowing the network to model complex relationships between inputs and outputs.

In the context of machine learning, we expand the computational graph G by integrating a special node dedicated to evaluating a loss function, denoted $\mathcal{L}$. This node ingests two sets of inputs: the result of all output nodes collected in the vector $\vec{o}$ and a predetermined vector $\vec{y} \in \mathbb{R}^{out}$, which is referred to as ground truth. The loss node's responsibility is to measure the graph's accuracy by contrasting its output vector $\vec{o}$ with the ground truth $\vec{y}$. Formally, we can express the loss function as a mapping $\mathcal{L} : \mathbb{R}^{out} \times \mathbb{R}^{out} \to \mathbb{R}$ that correlates the pairs of output and target vectors to a real number, signifying the magnitude of the loss denoted $\mathcal{L}(\vec{o}, \vec{y})$.

For ease of discussion, we introduce an equivalent representation of the computational graph. In this representation, edge weights are treated as distinct input nodes. Special computational nodes are introduced to perform multiplicative operations between these weight nodes and their associated variable nodes. Additionally, a unique computational node is introduced to compute the loss using values from all output nodes as illustrated in Figure 1.2. This expanded representation simplifies subsequent discussions on computing gradients during the training process, reducing them to a sequence of local computations between connected nodes.

1.1.2 *Neural Networks as Computational Graphs*

This section demonstrates that neural networks build on computational graphs. As we shift our terminology this observation clarifies: Henceforth, we will use the terms directed acyclic computational graph and artificial neural network and also the terms *vertices*, *neurons*, *gates*, *units* and *nodes* interchangeably. We denote the set of parameters or *weights* along the edges of the computational graph as Θ.

The input nodes transmit a *feature vector* $\vec{x}$. The feature vector represents the quantifiable properties, or *features*, of an object in a way that a computer can process. For instance, in image recognition, a feature vector might include the pixel values of an image. This vector provides the raw data that

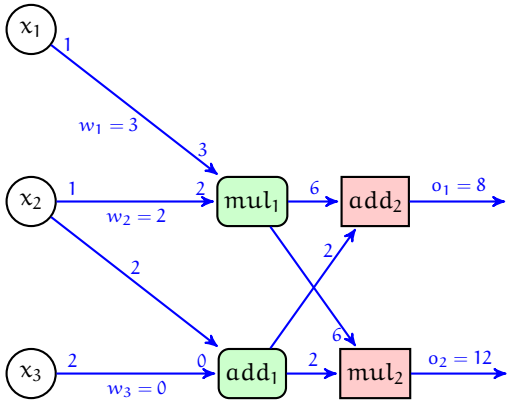

Figure 1.1: Illustration of a computational graph utilized in neural network architectures. The graph features three input nodes (white) and two computational nodes (green) which execute multiplication and addition operations respectively. The output of these computations is represented by two output nodes (red). Edges connecting the nodes are annotated with weights (e.g., $w_1 = 3$) and the computed values after each operation (e.g., $o_1 = 8$), guiding through the flow of calculations within the graph.

the network will use to learn and make predictions. Computational nodes within this network perform the main body of computation and are referred to as *hidden units*. The edges in neural networks are called *connections*.

The output nodes of the computational node store the vector $\vec{o}$ which is interpreted as a *prediction* or *target* in a neural network. This prediction serves different functions. In classification tasks, $\vec{o}$ often symbolizes a probability distribution over k classes. In regression tasks, $\vec{o}$ is generally real-valued.

Based on this terminology, we can articulate the goal of neural networks in a single sentence: We want to learn a function f that relates the input $\vec{x}$ to a predicted target $\vec{o}$ that is close to the ground truth $\vec{y}$ *even when the ground truth is not available*. This is achieved by iteratively prompting the neural network with *training examples* referred to as forward phase. A training example consists of a tuple $(\vec{x}, \vec{y})$ where $\vec{x}$ corresponds to the above-mentioned feature vector and $\vec{y}$ is the ground truth. By feeding $\vec{x}$ to our computational graph $\vec{o}$ is produced and subsequently evaluated against $\vec{y}$ using $\mathcal{L}$. In the context of neural networks the function evaluated in the loss node $\mathcal{L}$ is also call it the cost function, loss function, or error function.

The value of $\mathcal{L}$ is interpreted as an error signal used to adjust the network's parameters, or weights, denoted Θ. Thus, $\mathcal{L}$ should ideally be differentiable to facilitate neural network training using gradient descent. In a subsequent step, referred to as the backward phase, we cycle through the graph in reverse topological order and adapt the weights in a way that minimizes the loss using backpropagation, an efficient form of stochastic gradient descent (see Section 1.3. Note that minimizing the loss corresponds to maximizing the probability that the neural network prediction $\vec{o}$ matches the observed value $\vec{y}$. This reasoning leads us to an optimization problem, where we have some set of parameters Θ (the weights) that we manipulate in order to reduce the loss.

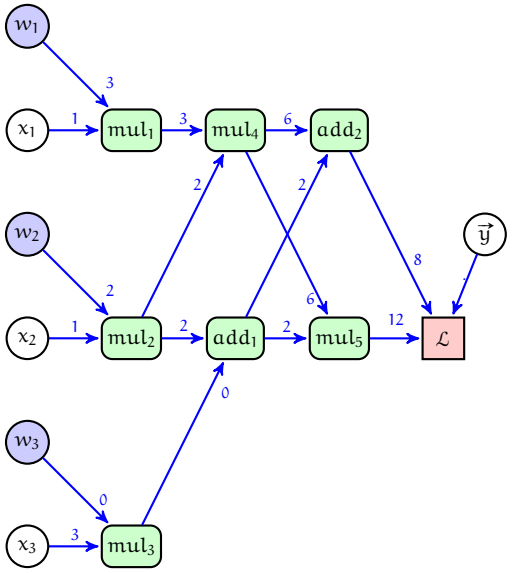

Figure 1.2: Depicted is a computational graph illustrating the flow of operations leading to the computation of the loss. The red node calculates the loss, incorporating both the result of preceding computations and the ground truth vector to evaluate the performance of the network. Weights are explicitly represented as separate nodes (violet).

1.1.3 *Single Layer Neural Network*

The preceding subsection gave a first overview of neural networks in the light of computational graphs. Next, we re-examine the basic concepts by studying the simplest neural network architecture possible, known as the *perceptron*, introduced by Rosenblatt [Ros58]. The input nodes to the perceptron are arranged in the input layer whereas the output layer contains a single computational node. All input nodes are connected to the output node. This setting is displayed in Figure 1.4. Since the input layer only transmits the data the perceptron is regarded as a *single-layer neural network*.

In Figure 1.4 the fixed input vector $\vec{x} = (x_1, x_2, x_3)^\intercal$ contains three so-called feature variables. The corresponding ground truth y is a scalar, taking values in $\{\pm 1\}$, and is known a priori during the training phase. The edges connecting the input and output layers in the figure are parameterized by the weights w_1, w_2, w_3 and collectively represented as a weight vector $\vec{w} = (w_1, w_2, w_3)^\intercal$.

In the perceptron model, the input feature variables are element-wise multiplied by the corresponding edge weights. These products are then summed to produce a single scalar value, to which a bias term $b \in \mathbb{R}$ is added. Subsequently, the sign function, acting as the activation function Φ, is applied to this sum. Formally, the output or prediction o of the perceptron is represented as:

$$o = \operatorname{sign}\{\vec{w}^\intercal \vec{x} + b\} = s \in \{\pm 1\} .$$

In Figure 1.4, the bias term is integrated into the network structure via an additional neuron in the input layer that outputs a constant value of 1. The

weight associated with this neuron effectively functions as the bias term. This representation choice allows for a more streamlined visual depiction of the model. To maintain this clarity, the bias term will not be explicitly shown in subsequent figures and discussions throughout the thesis. In the event of a mismatch between the predict target o and the ground truth y, the weight vector $\vec{w}$ is updated as follows:

$$\vec{w} \longleftarrow \vec{w} + \alpha(y - o)\vec{x}\,, \tag{1.1}$$

where the parameter $\alpha \in \mathbb{R}$ regulates the *learning rate of the neural network*. The learning rate controls the magnitude of weight adjustments in the model and is considered a *hyperparameter* (see Subsection 1.2.5 because it's set before training. During training, we repeatedly cycle through all the training examples in random order and iteratively adjust the weights until convergence is reached. Each complete pass through the training data is referred to as an epoch.

Remark 1 (Weight Update) The weight update of Equation 1.1 is a heuristic. In Section 1.1.2, we have already formulated the learning of neural networks as an optimization problem with the help of a loss function that the reader may have missed in the exposition above. In fact, later work (see [Bis95]) reverse-engineered the loss function that was implicitly used. Let $(\vec{x}_i, y_i)$ be the i-th training instance, then this training instance is mapped to the loss defined by $\mathcal{L} := \max\{-y_i(\vec{w}^\intercal \vec{x}_i), 0\}$. Note that the loss is non-negative and only positive if a mismatch is observed. Introducing matrix calculus notation we can calculate the derivative of the loss with respect to the weight vector

$$\frac{\partial \mathcal{L}}{\partial \vec{w}} = \left(\frac{\partial \mathcal{L}}{\partial w_1}, \cdots \frac{\partial \mathcal{L}}{\partial w_d}\right)^\intercal = \begin{cases} -y_i \vec{x}_i & \text{if } \operatorname{sign}\{\vec{w}^\intercal \vec{x}_i\} \neq y_i \\ 0 & \text{otherwise.} \end{cases}$$

Then, we apply gradient descent to minimize the loss. The idea is to take repeated steps in the opposite direction of the gradient at the current point because this is the direction of the steepest descent:

$$\vec{w} \leftarrow \vec{w} + \alpha y_i \vec{x}_i\,.$$

Remark 2 (Investigating the Bias) Let us briefly examine the purpose of the bias b. Essentially, the bias serves the role to account for possible imbalances in the class distribution that we want to predict. In many practical settings, it is the case that the n feature variables $\vec{x}_1, \vec{x}_2, \cdots$ are mean centered as a step of the preprocessing of the training data. Now, let us assume that the

sum of the ground truth values $y_i \in \{\pm 1\}$ is not 0. With $\vec{w}$ as the weight vector of the perceptron we derive:

$$\sum_{i=1}^{n} \vec{x}_i = 0 \neq \sum_{i=1}^{n} y_i$$
$$0 = \vec{w} \sum_{i=1}^{n} \vec{x}_i \neq \sum_{i=1}^{n} y_i$$

This discrepancy means that, without a bias term, the weight vector $\vec{w}$ alone cannot account for this imbalance to achieve zero training error. The weight vector is unable to shift the decision boundary away from the origin to correctly classify all training examples. Thus, the inclusion of a bias term is necessary for the perceptron to potentially attain zero training error in the presence of class imbalance. This logic expands to multi-layer neural networks where the incorporation of a bias significantly improves the model's accuracy [Agg23].

Remark 3 (Geometric Interpretation of Learning) A visual understanding of neural network learning is crucial. Assume that the input data is two dimensional with binary labels ± 1. Then, the perceptron learns a linear hyperplane defined by

$$\vec{w}^T \vec{x} = 0 \,, \tag{1.2}$$

where Equation 1.2 implicitly uses the feature engineering trick. In consequence, the data is labeled according to the site relative to the hyperplane (see Figure 1.3). With regard to Remark 2 we note that the bias corresponds to a translation of this hyperplane. This property solidifies our observation that the bias significantly improves the expressive capability of the perceptron because this translation cannot be simulated through a purely linear transformation. We call the data *linearly separable* if it is possible to learn an affine linear hyperplane that separates the two classes. This property is illustrated in Figure 1.3. In the case of linearly separable data, the perceptron always converges to provide zero error on the training data [Ros58]. However, if linear separability of the data is not guaranteed, we must find more powerful models which is the motivation of Subsection 1.1.4.

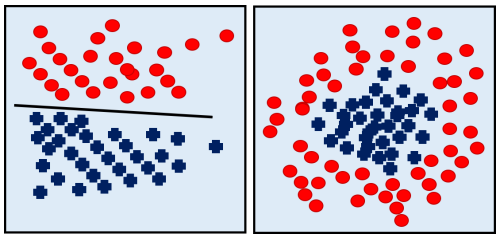

Figure 1.3: The data on the left is linearly separable. On the other hand, the data on the right has a pattern that cannot be discriminated by a linear hyperplane.

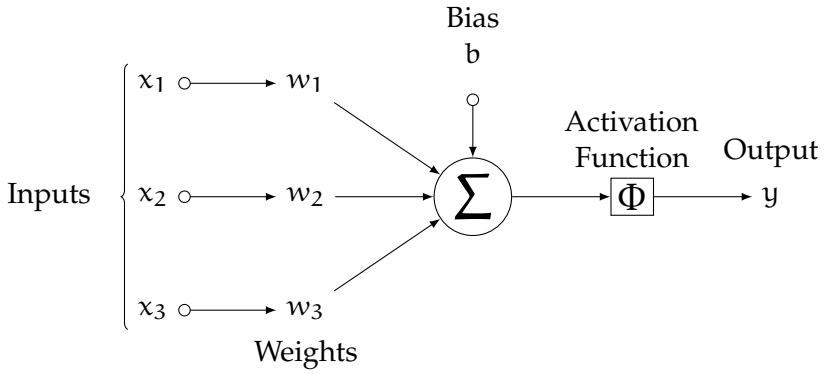

Figure 1.4: A single neuron called perceptron decomposed into a linear and a non-linear part. The Figure is adapted from [Agg23].

1.1.4 *Multi-Layer Neural Network*

The real power of neural networks is unleashed when the computational units are combined and stacked up to create deep networks. Then, the value computed by a single unit is forwarded to the nodes that can be reached through outgoing edges. This is achieved by arranging the input nodes in a single input layer, the nodes performing the main part of the computation in one or many hidden layers, and an output layer. In light of the computational graph G we can formulate this more rigorously. Let the set of nodes V of G be decomposed into a union of nonempty disjoint subsets, $V = \dot{\cup}_{l=0}^{L} V^l$ such that every edge in E connects some node in V^{l-1} to some node in V^l. In this vein, L is referred to as the depth of the network. The bottom subset V^0 is called the input layer. We denote the numbers of nodes $|V^i|$ in the i-th layer as p_i defining the dimension of a layer. Therefore, p_0 is the dimension of the input space and p_L is the dimension of the output.

Note that the first input layer is not counted. In Figure 1.5 all possible layer-to-layer connections are present. This property is called *fully connected* or *dense* [Agg23]. The triplet consisting of the set of nodes V, the edges E, and the activation function Φ determines the layer-wise structure of the neural network. From now on, we refer to this structure as the neural networks *architecture* denoted $\mathcal{A} = (V, E, \Phi)$ [SSS14].

Based on this reasoning, we offer a rigorous mathematical definition of multi-layer neural networks: We describe the cascading layer-wise computations of the graph G analytically as a nested compositional parameterized vector-to-vector function $f : \mathbb{R}^{p_0} \times \hat{\Theta} \to \mathbb{R}^{p_L}$ of the form

$$f(\vec{x}, \Theta) = \Phi_L \circ g_L \circ \cdots \circ \Phi_1 \circ g_1(\vec{x}, \Theta) \,,$$

where $\mathbb{R}^{p_0}$ defines the input space, g_l, with $g_l : \mathbb{R}^{p_{l-1}} \to \mathbb{R}^{p_l}$, is the linear pre-activation determined by the weights between the $(l-1)$th the lth layer, and Φ_l given by $\Phi_l : \mathbb{R} \to \mathbb{R}$ is the element-wise applied nonlinear activation function computed by the individual neurons in the lth layer [Mon+14]. The

parameter space $\hat{\Theta}$ is then the Cartesian product of the individual parameter spaces for each layer's weights defined as:

$$\hat{\Theta} = \hat{\Theta}_1 \times \cdots \times \hat{\Theta}_L$$

where each $\hat{\Theta}_l$ represents the space of all possible weight matrices for layer l and is defined as:

$$\hat{\Theta}_l = \mathbb{R}^{p_l \times p_{l-1}}$$

Therefore, a point $\Theta \in \hat{\Theta}$ represents a specific instantiation of the neural network's trainable parameters and is composed of the $p_1 \times p_0$ input matrix W_1, the $p_l \times p_{l-1}$ connection matrices W_l, and the $p_L \times p_{L-1}$ hidden to output matrix W_L. The dimensionaliyt of the parameter space is computed as follows:

$$\dim(\hat{\Theta}) = \sum_{l=1}^{L} (p_l \cdot p_{l-1}) \ .$$

Let us connect this formalism to the computational graph abstraction. In this vein, we make the following observations: The activation of the i-th neuron in the lth layer, denoted $h_l[i]$, is given by

$$h_l[i] = \Phi_l(g_l(\vec{h}_{l-1})[i]) \ , \tag{1.3}$$

where g_l is defined by

$$g_l(\vec{h}_{l-1}) = W_l \vec{h}_{l-1} \ . \tag{1.4}$$

Hence, the weights on the edges that start from the p^{l-1} neurons in the $(l-1)$th layer and are directed to the i-th neuron in the lth layer are stored in the p^{l-1} columns of the i-th row of the connection matrix, denoted $W_l[i,:]$. The output of the lth layer is a vector $\vec{h}_l \in \mathbb{R}^{p_l}$ of activations. By these definitions, the number of rows in the $p_l \times p_{l-1}$ connection matrix denotes the number of units in the lth layer, whereas the number of columns is determined by the number of computational units in the $(l-1)$th layer. With this reasoning, we have demonstrated the equivalence of the computational graph abstraction and the mathematical formalism presented in the present subsection.

Since we implicitly assume that we use the same activation function Φ throughout our network and furthermore apply Φ in an element-wise fashion we can simplify Equation 1.3 and Equation 1.4:

$$\begin{aligned}
\vec{h}_1 &= \Phi(W_1 \vec{x}) \text{ (Input to Hidden Layer)} \\
\vec{h}_l &= \Phi(W_l \vec{h}_{l-1}) \text{ (Hidden to Hidden Layers)} \\
\vec{o} &= \Phi(W_L \vec{h}_{L-1}) \text{ (Hidden to Output Layer.)}
\end{aligned}$$

To the confusion of many theorists and practitioners alike, in machine learning, there is a convention of transposing the *connection matrix* W_l and referring to $W_l^\intercal$ as the actual weight matrix.

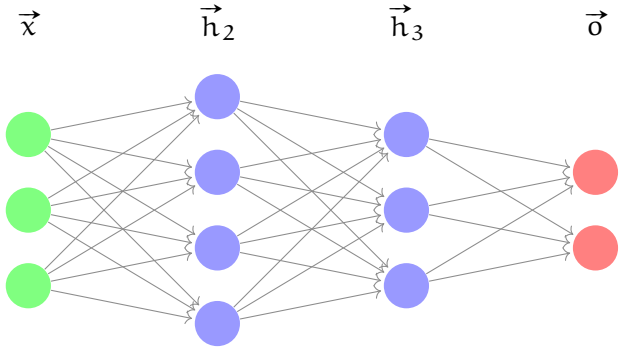

Figure 1.5: A Fully Connected Layer-Wise Structured Feed Forward Neural Network

1.2 NEURAL NETWORK COMPONENTS

In line with the chapter's incremental learning strategy, this section revisits the essential components of neural networks. We re-discover now-familiar concepts such as activation functions, output layers, loss functions, and hyperparameters, each underscored by examples and mathematical formalism. This section ends the introduction of neural networks and prepares the ground for future chapters where we discuss these components and their interactions in much more depth.

1.2.1 *Activation Functions*

A nonlinear activation function Φ is essential for highly expressive neural networks. Some of the most prominent activation functions are:

$$\begin{aligned}
\text{Sigmoid}(x) &= \frac{1}{1+\exp(-x)} \\
\text{Tanh}(x) &= \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)} \\
\text{Rectified Linear Unit}(x) &= \max(0,x) \\
\text{Leaky Rectified Linear Unit}(x) &= \begin{cases} x & \text{if } x \geqslant 0 \\ \epsilon x & \text{otherwise} \end{cases}
\end{aligned}$$

where the parameter $\epsilon > 0$ has to be specified before training or must be learned [Agg20]. In Figure 1.6 we observe that all functions are nonlinear but smooth in the sense that they are almost everywhere differentiable. In practice, critical points like $x = 0$ for the ReLU rarely become a problem because of the limited floated point precision in computers.

Note that functions like the `sigmoid` and `tanh` functions squash their input within a small interval. The gradients of those functions are largest near 0. For large negative or for large positive arguments these functions saturate meaning that a big increase in the absolute value of the argument

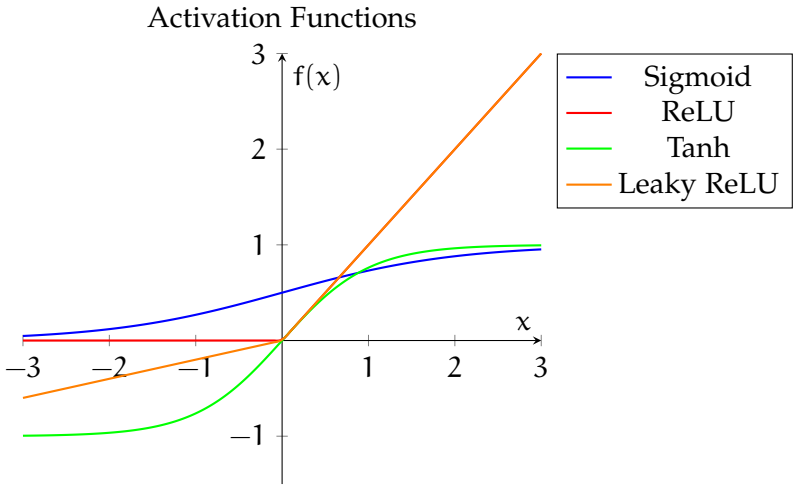

Figure 1.6: Examples of Classical Activation Functions

has minimal influence on the output value. This is reflected in the gradient of those functions that is close to 0 in these regions.

1.2.2 *The Power of Activation Functions*

In Remark 3 we saw that the perceptron faces a serious obstacle if the classes are not linearly separable. This subsection showcases how the expressive power of a neural network added by nonlinear activations mitigates this problem. The following example originates from [Agg23]. However, I have significantly expanded it to display the interplay between linear and nonlinear transformations. Suppose we have three data points at hand: $A = (-1, 1)$, $B = (0, 1)$ and $C = (1, 1)$. Furthermore, A and C are members of the same class and labeled with 1 whereas C is a member of its own class labeled with 0. The neural network consists of a single hidden layer with linear activation and a single output node O defined by

$$\begin{pmatrix} h_1 \\ h_2 \end{pmatrix} = \begin{pmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$
$$o = \begin{pmatrix} u_1 & u_2 \end{pmatrix} \begin{pmatrix} h_1 \\ h_2 \end{pmatrix}$$

It is a basic fact of linear algebra that affine linear transformations are only capable of shearing, translating, scaling, reflecting or rotating the space. Since these data points sit on a line, where C is in between A and B clearly there is no way an affine linear transformation can separate the data instances A

and B from C. Next, we add the ReLU activation to the network and set the weights of the matrix leading to the following situation:

$$\begin{pmatrix} \hat{h}_1 \\ \hat{h}_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ -1 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$
$$h_1 = \max\{h_1, 0\}$$
$$h_2 = \max\{h_2, 0\}$$
$$o = \begin{pmatrix} u_1 & u_2 \end{pmatrix} \begin{pmatrix} h_1 \\ h_2 \end{pmatrix}$$

In Figure 1.7 we observe that the matrix transformation corresponds to collapsing the 2-dimensional space into a single 1-dimensional line. Afterwards, the ReLU activation folds the space again mapping all data to the first quadrant of the coordinate system successfully rendering the two classes linearly separable. Setting $u_1 = 1, u_1$ we can predict training data with strictly positive output value with the class label 1 and data with non-positive output with the class label 0 enabling the network to classify the training examples with 0 loss.

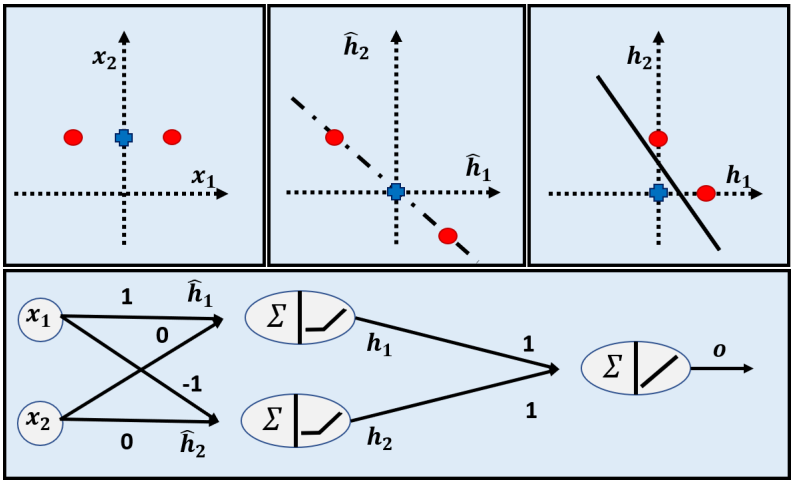

Figure 1.7: On the upper left part of the figure the data is not linearly separable. After the linear transformation, the data sits on the 1-dimensional subspace indicated by interspersed line. On the right, the data after the nonlinear activation is linearly separable. [Agg23].

1.2.3 *Output Layer*

The choice of the loss function is directly linked to the design of the output layer and depends on the application at hand [Agg23]. In this thesis, we restrict our focus to unordered multi-class prediction. In practical settings the input of the output layer corresponds already to the desired output dimension out $\in \mathbb{N}$. Therefore, assuming that an out-way classification is intended, we want the final output layer to map out real values into out

probabilities. Thus, every value of the output layer is interpreted as the neural network probability assigned to the i-th class. The preactivation values $\vec{v} \in \mathbb{R}^{o}ut$ that flow into every single node of the output layer are interpreted as real-valued *logits* that must be normalized to the range $[0, 1]$ in a way that they sum up to 1. In order to achieve this goal, the scalar *softmax* function $s_i : \mathbb{R}^{out} \to \mathbb{R}$ for the i-th output node is defined as

$$s_i(\vec{v}) = \frac{\exp v_i}{\sum_{j=1}^{k} \exp v_j} \, . \tag{1.5}$$

This function is applied at each neuron in parallel to compute the k postactivation values. In some contexts, it will also be convenient to denote the softmax as a vector-to-vector function $s : \mathbb{R}^{out} \to \mathbb{R}^{out}$.

$$s(\vec{v}) = (s_1(\vec{v}), \cdots, s_k(\vec{v})) \, . \tag{1.6}$$

The postactivation vector $\vec{s}$ of the softmax layer is used to compute the loss function. Using our notational convention, we see that $s_i(\vec{v}) = s(\vec{v})[i]$

1.2.4 *Loss Function*

In machine learning, we generally assume have independently identically distributed training instances (x, y) obtained from a probabilistic data generation source $q(x, y)$ that is considered as the ground truth. To formulate the concept of a loss function in a neural network in a rigorous manner, let's consider our function $f : \mathbb{R}^{in} \times \hat{\Theta} \to \mathbb{R}^{out}$ that represents the neural network.

We then define a loss function $\mathcal{L} : \mathbb{R}^{out} \times \mathbb{R}^{out} \to \mathbb{R}$ to measure the discrepancy between the predicted output $f(\vec{x}; \Theta)$ and the a priori observed ground truth $\vec{y}$. The role of $\mathcal{L}$ is to produce a scalar value that quantifies the error between the predicted and actual outputs for each input $\vec{x}$ given by $\mathcal{L}(f(\vec{x}; \Theta), \vec{y})$. The objective is to find Θ^* that minimizes the expected loss over the data generating distribution q:

$$\Theta^* = \arg\min_{\Theta \in \hat{\Theta}} \mathbb{E}_{(\vec{x}, \vec{y}) \sim q}[\mathcal{L}(f(\vec{x}; \Theta), \vec{y})] \tag{1.7}$$

In deep learning, the choice of loss function for prediction and classification tasks is grounded in information theory. Specifically, *cross-entropy* is widely used to measure the average number of bits needed to encode events z of an event space Ω drawn from a *ground truth distribution* q using the optimal code for an alternative probability distribution p defined over the same event space. Formally, cross-entropy between q and p is defined as:

$$H(q, p) = -\sum_{z \in \Omega} q(z) \log p(z) \tag{1.8}$$

In a neural network classification setting, the ground truth distribution q is usually represented as a one-hot encoded vector corresponding to the

actual class labels given in the training examples: Assuming k classes and a true class label with index i, $1 \leqslant i \leqslant k$, for the mth training instance we have $\vec{y}_m[i] = 1$ and $\vec{y}[j] = 0$ for $j \neq i$. Conversely, the predicted probabilities for each class, p, are generally produced by the softmax output layer of the neural network. Consequently, the machine learning adaptation of the cross-entropy function is given by:

$$\mathcal{L} = -\sum_{i=1}^{out} \vec{y}[i] \log(\vec{s}[i]) \, . \tag{1.9}$$

Here, $\vec{y}[i]$ (components of q) constitute the one-hot encoded vector, and $\vec{s}[i]$ (elements of p) denote the predicted probabilities for each class. This loss function quantifies the alignment between the model's predicted probabilities and the actual label distribution in the data set. We briefly mention how to interpret Equation 1.9 and refer to Subsection 2.2 where we discuss both views in much more detail: In terms of information theory, Equation 1.9 is referred to as the *log loss* which corresponds to the codelength needed to encode the event based on the neural network's prediction p that estimates the event's probability [Grü05]. Intuitively, the higher the probability of an event, the shorter the codelength we assign it to minimize the expected code length for sequences of events [Grü05]. The fundamental observation that probability distributions as *prediction strategies* are equivalent to *compressing schemes* is formalized by the *Kraft inequality* [McM56] that lies at the heart of machine learning theory. The other (more popular) perspective is rooted in the principle of maximum likelihood estimation: By minimizing the cross-entropy loss, we are searching for the set of parameters Θ^* that make the observed training instances the most probable under the distribution p that is modeled by f [GBC18].

1.2.5 *Hyperparameters*

Building on the principle of minimizing the cross-entropy loss to find the most probable set of parameters under the ground truth distribution, we introduce an additional layer of complexity: hyperparameters. These are not learned from the data but are set a priori. Some of the hyperparameters commonly adjusted in neural networks include the learning rate, the number of layers, their dimensionality, the number of epochs, and the activation function.

Formally, let $\hat{\eta}$ denote the hyperparameter space, and η be a specific hyperparameter configuration in $\hat{\eta}$. The space $\hat{\eta}$ is formally defined as the Cartesian product of the domain of each individual hyperparameter:

$$\hat{\eta} = \hat{\eta}_1 \times \hat{\eta}_2 \times \cdots \times \hat{\eta}_n$$

The optimization in neural network training can be viewed as a two-tier hierarchical procedure. At the inner level, for each fixed hyperparameter configuration η, the goal is to find $\Theta^*(\eta)$ given by Equation 1.7. On the outer

level, the hyperparameters η are optimized based on the performance of the best Θ obtained for each hyperparameter set:

$$\eta^* = \arg\min_{\eta \in \hat{\eta}} \mathbb{E}_{(\vec{x},\vec{y})\sim q}[\mathcal{L}(f(\vec{x}, \Theta^*(\eta)), \vec{y})]$$

This hierarchical optimization approach offers a structured framework for model training. The *inner loop* focuses on training the model to fit the data as closely as possible for a given set of hyperparameters, while the outer loop aims to find the hyperparameters that produce the best-generalized model. We are interested in implementing this hierarchical optimization efficiently. The implementation of the outer loop will be discussed in Section 3.1 whereas the implementation of the inner loop is the topic of Section 1.3.

1.3 FORWARD AND BACKWARD PROPAGATION

This section focuses on the *backpropagation algorithm*, a cornerstone in neural network training. Originally developed in 1970 under the name *reverse mode of automatic differentiation* [Lin70], its utility for neural networks was first recognized by Werbos [Wer74]. Much of deep learning's efficacy stems from the ability to automatically and parallelly learn the weights of extensive computational graphs, abstracting complex computations from the programmer [Agg23]. It is worth noting that Werbos's seminal thesis on backpropagation [Wer74] initially went unnoticed. Neural networks were largely considered untrainable [MP69] until the algorithm was rediscovered and brought into prominence by Rumelhart et al. in 1985 [RHW+85].

In Subsection 1.3.1, I present a brief general overview of neural network training process followed by an example where we backpropagate through a softmax layer. Having built an intuition we develop a more formal view on backpropagation in the last subsection.

1.3.1 *Forward-Backwards Loop in the Training Cycle*

Conceptually, neural network training can be thought of as a loop. To learn a trainable parameter $\theta \in \Theta$ of a neural network a training tuple $(\vec{x}, y)$ is randomly selected from the training data $\mathbb{X}^{train}$. Every loop consists of two phases: the *forward pass* and the *backward pass*:

1. In the *forward pass* the input $\vec{x}$ is forward-propagated through the network: according to the topological order of G all computational nodes are selected and evaluated successively. This ensures that the values of all incoming nodes have already been computed once we want to compute a nodes output. When this computation terminates, the loss function $\mathcal{L}$ is computed.

2. In the second phase we use gradient descent to update the current set of weights so that the loss decreases. This phase is called *backward pass*

because we calculate the gradients $\frac{\partial \mathcal{L}}{\partial \theta}$ for each weight $\theta \in \Theta$ in the *reversed topological order*. Then, we update all parameters:

$$\theta \leftarrow \theta - \alpha \frac{\partial \mathcal{L}}{\partial \theta} \, .$$

This process is repeated to convergence.

1.3.2 *Node-Wise Backpropagation*

This subsection presents the details of backpropagation. In technical terms, backpropagation is a recursive operation of the chain rule. Therefore, in order to understand the backward pass, we repeat two well-known results of calculus.

1. With the *univariate chain rule*

$$\frac{\partial f(g(x)}{\partial x} = \frac{\partial f(g(x)}{\partial g(x)} \frac{\partial g(x)}{\partial x} \tag{1.10}$$

   we obtain the derivative of a univariate composition function of the form $f \circ g(x)$.

2. The *multivariate chain rule* is a generalization of Equation 1.10:

$$\frac{\partial f(g_1(x), \ldots, g_k(x)}{\partial x} = \sum_{i=1}^{k} \frac{\partial f(g_1(x), \ldots g_k(x)}{\partial g_i(x)} \frac{\partial g_i(x)}{\partial x} \, . \tag{1.11}$$

   It is useful to visualize Rule 1.11 such that for any pair of source-sink nodes, the derivative of the variable in the sink node with respect to the variable in the source node is the sum of the expressions arising from the univariate chain rule being applied to all paths existing between any pair of nodes [Agg23].

Now, the introduction of neural networks as computational graphs pays off because this methodology greatly simplifies the exposition of backpropagation: Let us focus on a single neuron residing in a large computational graph. The neuron's operation denoted by $f : \mathbb{R} \to \mathbb{R}$ maps an input x and passes on the output $f(x) = z$ in the forward pass. Figure 1.8 translates Equation 1.10 to the computational graph abstraction. In the forward phase, the neuron outputs z which ultimately affects the network's loss $\mathcal{L}$. Furthermore, the neuron can already compute its *local gradient* $\frac{\partial z}{\partial x}$ because this operation is not dependent on other components of the graph. Once the loss is calculated, our single neuron receives information about its influence on the loss function during the backward pass stored in the *global gradient* $\frac{\partial L}{\partial z}$.

By Rule 1.10 we need to multiply the local gradient with the global gradient to compute $\frac{\partial L}{\partial x}$. This gradient flows further in a backward direction and informs other neurons of their influence on $\mathcal{L}$. This way of thinking about backpropagation emphasizes the recursive nature of the backward pass.

Analogously, in Figure 1.9 we transfer Equation 1.11 to our computational graph abstraction: In the forward pass, a local derivative is computed and stored. Furthermore, a copy of the output is passed along every outgoing edge to the neurons of the next layer. In the backward pass, a single global gradient is computed by summing all incoming gradients with respect to the neurons reachable through the outgoing edges. The global gradient with respect to the current neuron is scaled depending on the local gradient and sent further downstream.

When neural networks are understood in the light of computational graphs, it is straightforward to implement neural network training. Listing 1.1 provides a description of the computational graph object in which the overall forward-backward loop is implemented. In Listing 1.1 we note that every computational node used in the computational graph has to implement a forward and a backward call during forward and backward operation. An example of the behavior of a single computational node implementing the multiply operation is shown in Listing 1.2. Analogously, one can implement other computational nodes for any function.

In sum, we note: By recursively applying Rule 1.11, we propagate through a graph of arbitrary size and complexity in reverse topological order to compute all gradients.

Finally, I observe that each edge of the computational graph is traversed exactly twice (once in the forward pass and once in the backward pass) per training loop. This avoids an exponential running time with respect to graph size.

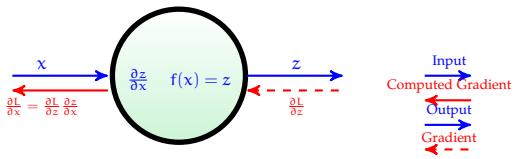

Figure 1.8: The blue arrows and formulas represent the forward pass where the input x is transformed to output z through function f. The red arrows and formulas depict the backward pass, indicating the propagation of the gradient $\frac{\partial \mathcal{L}}{\partial z}$ back through the node, used to compute the gradient with respect to the input $\frac{\partial L}{\partial x}$.

1.3.3 *Node-Wise Backpropagation through a Softmax*

The main point of backpropagation is that it allows for efficient calculation of the set of gradients for huge and complex graphs. To illustrate this point we are going to backpropagate through the softmax layer.

For the purpose of clarity in the presentation, we assume that the neural network first computes the multiply of a diagonal weight matrix $W^{\triangle} = \mathrm{diag}(w_1, \cdots, w_k) \in \mathbb{R}^{k \times k}$ with the fixed input $\vec{x} = (x_1, \cdots x_k)^T$. On the

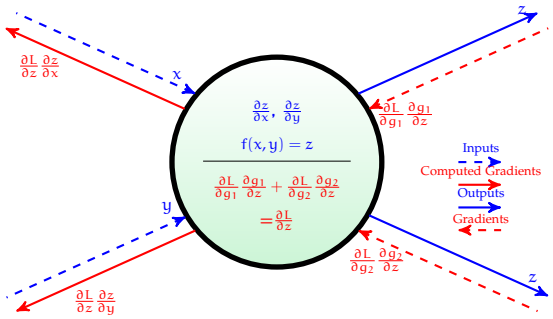

Figure 1.9: The blue elements are computed during the forward propagation through the network, while the red elements are computed in the backward pass, combining gradients from subsequent functions g_1 and g_2 with respect to z.

resulting vector, the scalar softmax s_i is applied. For the case $k = 3$ this procedure is depicted in Figure 1.10. We rewrite the scalar softmax s_i as:

$$s_i = \frac{\exp(x_i w_i)}{\sum_{j=1}^{k} \exp(x_j w_j)} .$$

We are interested in the derivative of s_i with respect to the weights: $\frac{\partial s_i}{\partial w_r}$ where $i, r \in \{1, 2, 3\}$.

For the softmax, there are two ways to compute these derivatives: The analytical approach directly implemented in the machine learning framework *PyTorch* [Vir+20] dramatically simplifies the backpropagation. However, this approach is not feasible when scaled to networks composed of hundreds of layers and billions of units. Thus, we solve the problem in the light of a computational graph which allows for automatic differentiation and is applicable in the general case.

1.3.3.1 *Node-wise Backpropagation: Analytical Approach*

For the analytical approach, there are two cases to consider:

1. When $i = r$, that is, the derivative of the softmax with respect to the weight in the numerator.
2. When $i \neq r$, that is, the derivative of the softmax with respect to the weight not in the numerator.

Let us handle these cases separately:

1. Case $i = r$: We will need to apply the quotient rule, which states that the derivative of $\frac{u}{v}$ is $\frac{vu' - uv'}{v^2}$.

Let $u = \exp(x_i w_i)$ and $v = \sum_{j=1}^{k} \exp(x_j w_j)$. The derivative of u with respect to w_i is $x_i \exp(x_i w_i)$, and the derivative of v with respect to w_i is $x_i \exp(x_i w_i)$. Substituting these into the quotient rule yields us:

$$\frac{x_i \exp(x_i w_i) \left(\sum_{j=1}^{k} x_j \exp(x_j w_j)\right) - \exp(x_i w_i) \left(x_i \exp(x_i w_i)\right)}{\left(\sum_{j=1}^{k} \exp(x_j w_j)\right)^2}$$

which simplifies to

$$s_i \left(x_i - s_i x_i\right)$$

or equivalently

$$s_i x_i (1 - s_i) \ .$$

2. Case $i \neq r$: In this case, the derivative of u with respect to w_r is 0, and the derivative of v with respect to w_r is $x_r \exp(x_r w_r)$. Substituting these into the quotient yields:

$$\frac{\partial s_i}{\partial w_r} = \frac{0 - uv'}{v^2}$$

which translates to

$$\frac{-\exp(x_i w_i) \left(x_r \exp(x_r w_r)\right)}{\left(\sum_{j=1}^{k} \exp(x_j w_j)\right)^2}$$

which further simplifies to

$$-s_i s_r x_r$$

In conclusion, the derivative of the scalar softmax function s_i with respect to the weights w_r is:

$$\frac{\partial s_i}{\partial w_r} = \begin{cases} s_i x_i (1 - s_i) & \text{if } i = r \\ -s_i s_r x_r & \text{if } i \neq r \end{cases} \tag{1.12}$$

This concludes the derivation of the derivative of the softmax function with respect to the weights w_r.

1.3.3.2 *Node-Wise Backpropagation: Computational Graph Abstraction*

When confronted with huge neural networks, it is crucial to understand the fundamental mechanics of backpropagation in the context of computational graphs. This is why we will illustrate the forward and backward passes through the softmax layer implemented as a computational graph depicted in Figure 1.10. Before we work through this concrete example we initialize our weight vector $\vec{w} = (w_1, w_1, w_2)^\intercal = (0.5, 1, -0.5)^\intercal$ and our input vector $\vec{x} = (x_1, x_2, x_3)^\intercal = (1, -2, 3)^\intercal$.

The forward pass is calculated step by step in the Table 1.1. Note that the forward propagated values and the local gradients of every node in the graph are computed in topological order. In the backward pass, we use the chain rule to compute the derivatives of the loss with respect to the weights. We always initialize the global gradient:

$$\frac{\partial out}{\partial out} \stackrel{\text{def}}{=} \lim_{h \to 0} \frac{(out + h) - out}{h} = 1 \ .$$

In Table 1.2 we see how all gradients are calculated in the reversed topological order of the computational graph.

It is always advised to verify the correctness of our computation by comparing our analytical result from Formula 1.12 with the result in Table 1.2:

$$\begin{aligned}
\frac{\partial s_3}{\partial w_1} &= -s_3 \cdot s_1 \cdot x_1 \\
&= -0.1111656223 \cdot \frac{e^{0.5}}{e^{0.5} + e^{-2} + e^{-1.5}} \approx -0.09131244481 \\
\frac{\partial s_3}{\partial w_2} &= -s_3 \cdot s_1 \cdot x_2 \\
&= -0.1111656223 \cdot \frac{e^{-2}}{e^{0.5} + e^{-2} + e^{-1.5}} \cdot (-2) \approx 0.01499076381 \\
\frac{\partial s_3}{\partial w_3} &= s_3 \cdot (1 - s_3) \cdot x_3 \\
&= 0.1111656223 \cdot (1 - 0.1111656223) \cdot 3 \approx 0.2964234802
\end{aligned}$$

I claimed that the graph abstraction offers the practitioner a direct approach for implementing the backpropagation algorithm. Listing 1.1 delineates the data structure for the computational graph that stores and uses computational nodes for forward and backwards passes through the network. Every node-operation is implemented as its own class and has to implement the forward and the backward operation. For instance, Listing 1.2 outlines the implementation of a multiply operation. By implementing all necessary operations one can construct a fully functional neural network training loop by using only basic python code.

Observation 1 (Interpreting Gradient Flow) Note again how Figure 1.10 highlights the dynamic flow of gradients. It is noteworthy to mention that the add gate functions as an effective distributor, equally partitioning the gradient flow to all preceding connected nodes. With this in mind, it is easy to derive a similar characteristic for a *max* gate. During forward propagation, exclusively the value from the node with the maximum value is allowed to pass through this gate. Consequently, during backpropagation, the gradient flow is solely directed to this particular node.

Furthermore, it's crucial to understand that the operations performed at these gates are not fixed and can be arbitrarily chosen. For instance, the entire graph in 1.10 could be condensed into a single softmax gate. I made

the decision to choose very basic gates to illustrate the power of partitioning the problem in pieces when dealing with complex systems in order to understand larger, more intricate networks.

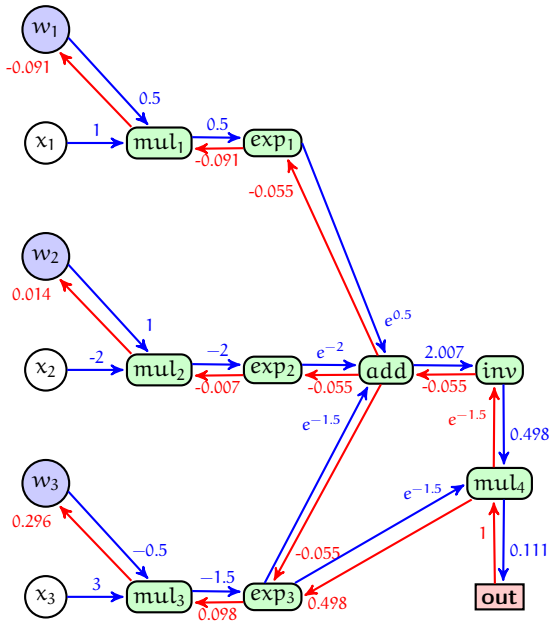

Figure 1.10: The diagram illustrates a part of a computational graph implementing a softmax function, showcasing the forward and backward passes essential for neural network training. Each node represents a mathematical operation, and the edges indicate the flow of data and gradients.

1.3.4 *Layer-Wise Backpropagation in Neural Networks*

So far, we have restricted ourselves to calculating scalar-valued node-to-node derivatives in computational graphs. To accelerate backpropagation with the help of Graphics Processor Units which are particularly efficient with vectors, matrices and tensors, backpropagation is performed in a layer-wise fashion [Agg23]. We restrict our analysis to neural networks consisting of alternately arranged linear layers followed by activation layers. Let us suppose that the connection matrix W denotes the transformation matrix between the i-th and the $(i+1)$th layer. In the forward pass the output vector $\vec{z}_i$ is transformed and forward propagated as follows:

$$\vec{z}_{i+1} = W\vec{z}_i \,. \tag{1.13}$$

If $\vec{g}_{i+1}$ denotes the gradient of the loss with respect to $\vec{z}_{i+1}$ we backpropagate:

$$\vec{g}_i = W^\intercal \vec{g}_{i+1} \,. \tag{1.14}$$

According to the higher dimensional chain rule [Spi18] the gradient is the transpose of the derivative of the output in terms of the input, so the weight matrix W in Equation 1.14 is transposed. For the second case we assume that

Functions	Forward Propagated Value	Stored Local Gradients
$mul_1(w_1, x_1)$	$w_1 x_1 = 0.5$	$\frac{\partial mul_1}{\partial w_1} = x_1 = 1$
$mul_2(w_2, x_2)$	$w_2 x_2 = 2$	$\frac{\partial mul_2}{\partial w_2} = x_2 = -2$
$mul_3(w_3, x_3)$	$w_3 x_3 = -1.5$	$\frac{\partial mul_3}{\partial w_3} = x_3 = 3$
$exp_1(mul_1)$	$\exp mul_1 = e^{0.5}$	$\frac{\partial exp_1}{\partial mul_1} = exp(mul_1) = e^{0.5}$
$exp_2(mul_2)$	$\exp mul_2 = e^{-2}$	$\frac{\partial exp_2}{\partial mul_2} = exp(mul_2) = e^{-2}$
$exp_3(mul_3)$	$\exp mul_3 = e^{-1.5}$	$\frac{\partial exp_3}{\partial mul_3} = exp(mul_3) = e^{-1.5}$
$add(exp_1, exp_2, exp_3)$	$exp_1 + exp_2 + exp_3$ ≈ 2.007186714	$\frac{\partial add}{\partial exp_1} = \frac{\partial add}{\partial exp_2} = \frac{\partial add}{\partial exp_3} = 1$
$inv(add)$	$1 \div add \approx 0.4982097545$	$\frac{\partial inv}{\partial add} = -(add)^{-2}$ ≈ -0.2482129595
$mul_4(inv, exp3)$	$inv \cdot exp_3 \approx 0.1111656223$	$\frac{\partial mul_4}{\partial inv} = e^{-1.5}$ $\frac{\partial mul_4}{\partial exp_3} = inv$
$out(mul_4)$	mul_4	$\frac{\partial out}{\partial mul_4} = 1$

Table 1.1: Forward Pass in a Computational Graph

Node Gradient	Chain Rule	Gradient Value
$\frac{\partial out}{\partial mul_4}$	$\frac{\partial out}{\partial out}\frac{\partial out}{\partial mul_4}$	$1 \cdot 1 = 1$
$\frac{\partial out}{\partial inv}$	$\frac{\partial out}{\partial mul_4}\frac{\partial mul_4}{\partial inv}$	$1 \cdot e^{-1.5} = e^{-1.5}$
$\frac{\partial out}{\partial exp_3}$	$\frac{\partial out}{\partial mul_4}\frac{\partial mul_4}{\partial exp_3}$	$1 \cdot inv \approx 0.4982097545$
$\frac{\partial out}{\partial add}$	$\frac{\partial out}{\partial inv}\frac{\partial inv}{\partial add}$	$e^{-1.5} \cdot 0.2482129595 \approx -0.0553837974$
$\frac{\partial out}{\partial exp_1}$	$\frac{\partial out}{\partial add}\frac{\partial add}{\partial exp_1}$	$-0.0553837974 \cdot 1 = -0.0553837974$
$\frac{\partial out}{\partial exp_2}$	$\frac{\partial out}{\partial add}\frac{\partial add}{\partial exp_2}$	$-0.0553837974 \cdot 1 = -0.0553837974$
$\frac{\partial out}{\partial exp_3}$	$\overset{+}{=} \frac{\partial out}{\partial add}\frac{\partial add}{\partial exp_3}$	$+= -0.0553837974 \cdot 1 = 0.4428259571$
$\frac{\partial out}{\partial mul_1}$	$\frac{\partial out}{\partial exp_1}\frac{\partial exp_1}{\partial mul_1}$	$-0.0553837974 \cdot e^{0.5} = -0.09131244483$
$\frac{\partial out}{\partial mul_2}$	$\frac{\partial out}{\partial exp_2}\frac{\partial exp_2}{\partial mul_2}$	$-0.0553837974 \cdot e^{-2} = -0.007495381$
$\frac{\partial out}{\partial mul_3}$	$\frac{\partial out}{\partial exp_3}\frac{\partial exp_3}{\partial mul_3}$	$0.4428259571 \cdot e^{-1.5} = 0.09880782673$
$\frac{\partial out}{\partial w_1}$	$\frac{\partial out}{\partial mul_1}\frac{\partial mul_1}{\partial w_1}$	$-0.09131244483 \cdot 1 = -0.09131244483$
$\frac{\partial out}{\partial w_2}$	$\frac{\partial out}{\partial mul_2}\frac{\partial mul_2}{\partial w_2}$	$-0.007495381 \cdot -2 = 0.014990762$
$\frac{\partial out}{\partial w_3}$	$\frac{\partial out}{\partial mul_3}\frac{\partial mul_3}{\partial w_3}$	$0.09880782673 \cdot 3 = 0.2963148804$

Table 1.2: Backward Pass in a Computational Graph

Listing 1.1: Implementing Forward and Backward Phase in a Computational Graph

```
class NeuralNetwork(object):
    def forward(inputs):
        # sort computational graph in topological order
        self.graph.nodes_sort_topological()
        # Set the value of input nodes to the corresponding value in the
             input vector
        for i, node in enumerate(self.graph.input_nodes):
        node.value = inputs[i]
        # Traverse the graph. For each node, perform the forward
            operation,
        for node in self.graph:
            node.forward()
        #The final node in the graph computes the loss
        loss = self.graph[-1].value
        return loss

    def backward():
        # Clear gradients from previous iteration
        self.graph.clear_gradients()
        # traverse graph in topologically reverse order
        reverse(self.graph)
        for node in self.graph:
            node.backward()
        # Extract gradients for all parameters into a single vector
        gradients = self.graph.get_gradients()
        return gradients
```

Listing 1.2: Neural Network Training in Pseudo-Code

```
class MultiplyNode(object):
    def forward(x,y):
        #compute output
        z= x*y
        #store local gradient
        self.x = x
        self.y =y
        return z

    def backward(dz):
        # Compute gradient with respect to x
        dx = self.y * dz
        # Compute gradient with respect to y
        dy = self.y * dz
        return [dx, dy]
```

the activation function Φ is applied to each coordinate of the output vector of the i-th layer to obtain the activation of the $(i+1)$th layer. Then, forward propagation between these layers is defined by:

$$\vec{z}_{i+1} = \Phi(\vec{z}_i) \, ,$$

where we use our already introduced notation denoting element-wise application of Φ. Analogously, for the backward pass we apply the derivative Φ' in an element-wise fashion to the vector argument:

$$\vec{g}_i = \vec{g}_{i+1} \odot \Phi'(\vec{z}_{i+1}), \tag{1.15}$$

where $\odot$ denotes element-wise multiplication [Agg23]. The exact derivations of Equation 1.14 and Equation 1.15 can be found in [Agg20].

Until now we have only obtained the loss-to-node derivatives but we actually need the *loss-to-weight-derivatives* to perform the gradient descent update. The gradient of the weight between the rth unit of the $(i-1)$th layer and the qth unit of the i-th layer is obtained by multiplying the rth element of z_{i-1} with the qth element of $\vec{g}_i$ [Agg23]. A trick for efficiency is to compute the outer product of $\vec{g}_i$ and $\vec{z}_{i-1}$ yielding the entire matrix M of derivatives of the loss with respect to the elements of the weight matrix between the $(i-1)$th layer and the i-th layer:

$$M = \vec{g}_i \vec{z}_{i-1}^{\,T} \, .$$

The method of computing the matrix M through the outer product, as opposed to calculating each derivative individually, capitalizes on the parallel processing capabilities of modern GPUs. In conclusion, decoupling the linear from the nonlinear activations makes backpropagation simple, automatable, and efficient.

1.3.5 *Revisiting Activation Functions in the Light of Backpropagation*

Following the introduction of the backpropagation algorithm in the previous section, we re-examine the specific properties of activation functions with regard to their suitability for gradient flow.

1.3.5.1 *Sigmoid Function*

The `sigmoid` function was used in the early days of neural networks since it was interpretable as a saturating firing rate of a neuron [HM95]. Furthermore, activation functions that squash any arbitrarily ranged input into the interval $[0, 1]$ can be interpreted as calculating an output probability and are helpful in constructing loss functions [Agg23]. An undesirable property of the sigmoid function is that it tends to lead to vanishing gradient flow (discussed in detail in Subsection 2.3.2.5). That is, when the neuron's activation saturates at either tail of 0 or 1, the gradient at these regions is almost zero. An even more important effect is that the derivative of the sigmoid function

is *always* smaller than 1. In this context, recall that during backpropagation, local gradient will be multiplied with the gradient of this gate's output with respect to the loss function. In neural networks we stack many layers, multiplying these gradients. The product of many smaller than 1 values quickly converges to zero. In consequence, this activation has fallen out of favor in recent years.

1.3.5.2 *Tanh Function*

The tanh function is a scaled sigmoid, in particular: $\tanh(x) = 2\sigma(2x) - 1$. Hence, the tanh nonlinearity squashes a real-valued number to the range $[-1, 1]$. Like the sigmoid neuron, its activations saturate, but unlike the sigmoid neuron its output is *zero-centered* and symmetric. It has been argued that the sigmoid's non-zero mean slows down gradient descent [LeC+02].

Additionally, random initial weights in a neural network are typically not predictive. Note that in many cases neural network weights are initialized with randomly drawn mean-centered values inducing small inputs into the activation function. In this context, the tanh function has a high gradient at inputs near 0 which it maps to 0. The combination of zero-centered initialization with a zero-centered activation has been found to be desirable at the beginning of backpropagation [GB10]. This observation is corroborated by evidence that deep networks with sigmoids but initialized from unsupervised pre-training do not suffer from this behavior [GB10]. In general, tanh activation is preferred over the sigmoid function [GB10].

1.3.5.3 *ReLU Function*

Rectifier-based functions like ReLU and leaky ReLU [MHN+13] have become popular in the last few years and the de-facto standard in neural networks [KSH12]. There are several positive features:

1. Compared to classic activation functions that involve expensive operations, the ReLU has low computational cost. In this vein, ReLU was found to greatly accelerate the convergence of stochastic gradient descent compared to the sigmoid and tanh functions [KSH12].

2. ReLU units increase the sparsity in the neural networks activations as the percentage of neurons that are active at the same time is low [Api+21]. There is empirical evidence that sparse representations seem to be more beneficial than dense representations [MTP15]. In this context, ReLU networks promote uniform sparsity across hidden units, fostering rich and varied internal representations that enhance learning. [MHN+13].

On the other side, we can identify several disadvantages with the ReLU unit:

1. Whereas the sigmoid effectively prevents activations from blowing up by squashing them in a small interval such a mechanism does not exist in the positive direction for the ReLU.

2. In the literature, the phenomenon of *dead ReLU neurons* is well-known [Dat20]. It has been observed that the gradient flowing through a ReLU neuron could cause the weights right before the ReLU to update in such a way that the neuron will never activate on any data point again. Specifically, if the inputs to this neuron are always non-negative whereas all the weights have been initialized through random initialization or are updated to be negative, the output of the ReLU will always be 0. If this form of saturation persists over all training data points for a specific neuron, the gradient of the loss with respect to the weights just before this neuron will always be zero. Since the weights of this neuron will be never updated during training this neuron can be considered dead. The dying ReLU problem occurs often in settings where the learning rate is set too high [Api+21]. The motivation for the introduction of the leaky ReLU [MHN+13] was to mitigate the issue of dead neurons. By allowing a small, non-zero gradient for negative inputs, the leaky ReLU ensures that neurons remain always active.

2

THEORETICAL INSIGHTS INTO LARGE LANGUAGE MODELS

Many algorithmic advancements covered in Chapter 3 aren't rooted in particularly rigorous theory. Despite the lack of theoretical foundation, this hands-on mentality dominates current research in deep learning because of its success to solve real-world problems whereas other approaches like *Bayesian networks* [CP12] or *support vector machines* [Nobo6] moved into the background even though they were more accessible to mathematical analysis. Despite this trend, understanding theoretical underpinnings of neural networks is not only of academic interest but is crucial for advancing the field because these insights can guide us towards new breakthroughs. In this vein, recent research efforts on theoretical properties of neural networks continued and have found partial answers to fundamental questions: In hindsight, it is easy to take it for granted that computers in general and neural networks in specific can learn. Nonetheless, as mentioned in the beginning of Section 1.3 this realization remained unclear for many decades. Thus, in Section 2.2 we attend to the question, *why do neural networks work at all*? In Chapter 1 we faced the technical details of tinkering neural networks to make them somehow work. In this context, two further questions arise: Is it worth the effort, or in other words, *how powerful are these neural networks*? Second, *what happens inside those intricate networks*? We deal with both questions in the final Section 2.3. However, the most basic questions that current textbooks [Agg20] [Agg23] [Bis95] [GBC18] do not approach in a straightforward and precise manner is: *What actually are those large language models everybody talks about?* Section 2.1 answers this.

2.1 LARGE LANGUAGE MODELS

Until the recent success of the generative pre-trained transformer series (GPT-n) that started at around 2017 the dominant approach to creating a machine learning system for *question answering*, *machine translation* and *reading comprehension* was to collect data that included a description of the desired task and the correct behavior for the task, to train the system to imitate these behaviors and afterward evaluate it on a test set [Rad+19] [Amo+16]. This approach is referred to as *supervised learning*. However, this methodology was limited to extremely narrow tasks and even there displayed regularly erratic behavior due to the diversity and variety of possible inputs [JL17]. The pivotal insight by a research group from OpenAI in 2019 [Rad+19] was that it is possible to teach large language models language processing tasks in an *unsupervised way* thus doing away with the scaling problem associated with the creation of supervised learning sets. In Subsection 2.1.1 we introduce the

unsupervised learning paradigm which corresponds essentially to training a statistical model [MD91]. Afterward, we walk through the seminal large language model proposed by Bengio [Ben+03] from language preprocessing (Subsection 2.1.2), through neural networks as statistical language models (Subsection 2.1.3), to their evaluation (Subsection 2.1.4).

2.1.1 *Statistical Language Model*

The implicit assumption maintained throughout this thesis is that sequences of language tokens such as characters, subwords, or words are independently and identically sampled from an infinite data source, denoted q. We suppose that q is not uniformly distributed, but, in contrast, there are interesting patterns in these sequences that we want to learn through a statistical language model, denoted p. Learning these patterns from the observation of unlabeled sequences falls under the unsupervised learning paradigm because no human intervention is necessary. Definition 3 makes this more precise:

Definition 3 (Statistical Language Model) Let us define the vocabulary V to be the finite set of all potential language tokens we want to model. Then, given a set of tokens $\{w_1, w_2, \cdots, w_m\}$ where $w_i \in V$ for all $1 \leqslant i \leqslant m$ sampled from an independently and identically distributed *discrete probability mass function* q, a statistical language model p is a *discrete probability mass function* defined over the power set of V, denoted 2^V. The goal of the statistical language model p is to approximate the true joint distribution q: $p(w_1, \cdots, w_m) \approx q(w_1, \cdots, w_m)$.

Remark 4 (Probabilistic Prediction Strategy) Since language has a natural sequential ordering we factorize p as the product of conditional probabilities given by

$$p(w_1, \cdots, w_m) = \prod_{i=1}^{m} \frac{p(w_1, \cdots, w_{i-1}, w_i)}{p(w_1, \cdots, w_{i-1})} = \prod_{i=1}^{m} p(w_i | w_1, \cdots, w_{i-1}) \ . \quad (2.1)$$

Equation 2.1 admits a fundamental re-interpretation of probability distributions as forecasting systems, mapping each individual sequence of *past observations* $w_1, \cdots, w_{i-1}$ to a probabilistic prediction of the next outcome [Daw84]. In reverse 2.1 indicates that any probabilistic prediction strategy for the sequential prediction of m outcomes may be considered a probabilistic distribution of the whole sequence $w_1, \cdots, w_m$. In practice, p is assumed to possess the *Markov property*:

$$p(w_i | w_1, \cdots w_{i-1}) = p(w_i | w_{i-n}, \cdots, w_{i-1}) \ .$$

where n constitutes the context length for our language model p. The intuition is that only a restricted number of previous tokens is informative when predicting the next token.

2.1.1.1 *The Curse of Dimensionality*

To be able to appreciate the magic of neural networks as statistical language model learners we must first examine the difficulty of modeling a joint distribution of discrete random variables in general: This difficulty is derived by the *curse of dimensionality*. Let us assume we would like to learn the joint distribution of ten arbitrary consecutive words in a natural language with a vocabulary of size 200.000. Then, there are 200.000^{10} parameters to adjust [Ben+03]. Thus, directly modeling the prediction function, as in *word n-gram statistical models* [NW96], becomes quickly intractable due to its exponential space complexity with regard to context length.

2.1.1.2 *Building an Intuition for Generalization in Discrete High Dimensional Spaces*

Another obstacle arises from the discrete nature of language: Unlike in the case of continuous data it is hard to generalize from observed sequences of language tokens because a change of a single discrete variable might have a drastic impact on the value of the function that we want to estimate [Ben+03]: Let us assume that our tokens are characters, and we model English text. Then, given the sequence 'h','i','s' we want our model to predict the *space character* with high probability because his is a prevalent word. However, changing a single variable, 'h','o','s', the space character should have zero probability because hos does not constitute a word.

In order to build an intuition what we are trying to do with a large language model we visualize how generalization works in a high-dimensional discrete space: Here, we need to distribute probability mass, initially focused on training points, in a non-uniform manner along *significant high-dimensional trajectories*. Take the sentences:

A mouse is racing in the room
The child is scrabbling in a room.

Here, "mouse" and "child" serve similar roles; they both denote entities capable of movement. Similarly, "room" is contextually linked to the articles "the" and "a". For a statistical language model, it's important to recognize these semantic and syntactic connections, thereby allowing for the transfer of probability mass. For instance, with the fragment *is racing in a*, the model should predict "room" with high probability, despite only being trained *on racing in the room*. In consquence, we would like our neural network to learn similar real-valued, high-dimensional vectors for words that encapsulate the semantics semantics and syntax. For instance, if the model frequently encounters "mouse" and "cat" in contexts of movement, it should assign these words vectors that are close in space, indicated by a high dot product. In essence, we counter the curse of dimensionality by learning exponentially, using the continuity of real-valued vectors that embody the distributed features of words [Ben+03].

2.1.1.3 *Overcoming the Curse of Dimensionality*

The strategy proposed by Bengio et al. [Ben+03] to tackle the curse of dimensionality involves utilizing a neural network to learn the statistical language model as well as the feature representations in a data-driven fashion. The procedure can be outlined as follows:

1. Associate each token in the language with a real-valued, high-dimensional vector known as an embedding vector.
2. Employ a multi-layer neural network to serve as the statistical language model, denoted by p, which predicts subsequent tokens.
3. Learn both the embedding vectors and the parameters of p simultaneously through backpropagation, driven by data.
4. Adjust the model's parameters iteratively by training on a substantial number of examples, refining the model's performance.

2.1.2 *Language Preprocessing*

Having discussed the general idea, we walk through a language model and start with language preprocessing: the sequence of tokens must be transformed into a vector representation suitable to the input of a neural network. This work is divided into two tasks: *vocabulary building* and *data set building*.

2.1.2.1 *Building a Vocabulary*

As a first step of preprocessing we perform *tokenization* [Agg18]. Tokenization refers to how a piece of text is represented as a sequence of vocabulary elements. There are several ways to process the text, extract its vocabulary V and represent it as a sequence of tokens:

1. One possible choice is to define V as the set of alphabet characters plus punctuation. The text *Hello!* would be a sequence of length six represented as $['\mathtt{h}', '\mathtt{e}', '\mathtt{l}', '\mathtt{l}', '\mathtt{o}', '!']$. A disadvantage is that single characters are too basic units to carry substantial semantic meaning.
2. Alternatively, V consists of all words of a language plus punctuation. For instance, the text *Hello!* would be a sequence of length two: $['\mathtt{Hello}', '!']$. An apparent disadvantage is that the sizes of such vocabularies easily exceed 200.000 entries. Additionally, a model cannot deal with words not encountered during training.
3. The third favored method is subword tokenization: V is represented as a set of commonly occurring word segments like *'under'*, *'ed'*, *'to'*. Frequent words like 'the' are also tokens. In order to include all possible words, alphabet characters are also incorporated. Hence, neural networks trained with sub-word tokenization can deal with arbitrary

segments of text. Sub-word tokenization is not trivial. A powerful sub-word tokenization algorithm is *byte pair encoding* [Shi+99] used in the GPT-n series. As a reference, in GPT-2 [Rad+19] and GPT-3 [Bro+20] the vocabulary includes 50.257 unique sub-words, a substantial reduction with regard to the size of word-derived vocabularies.

In addition to the tokens determined by the tokenization algorithm, a number of special tokens are added to the vocabulary. Typically, a token for the start and end of a sequence, and an 'out-of-vocabulary' token guide the language model. Furthermore, for simpler models other pre-processing steps are possible such as lower-casing or filtering less frequent tokens to reduce the vocabulary size [Rad+19].

There are two ways to numerically represent the a language token to the neural network. One way is to assign each token a unique index ranging from 1 to $|V|$, the ordinality of the vocabulary. Another common approach is *one-hot encoding* where the j-th token is represented as a (very!) high-dimensional vector $\vec{v}_j \in \mathbb{R}^{|V|}$ which contains a single 1 at the j-th position and is everywhere else 0. Table 2.1 shows both encodings for a small vocabulary of eight words.

Word	Index Encoding	One-Hot Encoding
shall	1	$(1,0,0,0,0,0,0,0)^T$
I	2	$(0,1,0,0,0,0,0,0)^T$
compare	3	$(0,0,1,0,0,0,0,0)^T$
thee	4	$(0,0,0,1,0,0,0,0)^T$
to	5	$(0,0,0,0,1,0,0,0)^T$
a	6	$(0,0,0,0,0,1,0,0)^T$
summer	7	$(0,0,0,0,0,0,1,0)^T$
day	8	$(0,0,0,0,0,0,0,1)^T$

Table 2.1: Indexed and one-hot encoded vectors for a vocabulary of size eight

2.1.2.2 *Building the Data Set*

In general, the structure of the data set depends on the task at hand. Since our model tries to predict a word based on the previous context our data set features tuples of context-target pairings. For instance, given the sequence *Italian cuisine is the best in the world.* and a context size of three the data set contains:

([italian, cuisine, is], the)
([cuisine, is, the], best)
([is, the, best], in)
([the, best, in], the)

([best, in, the], world)
([in, the, world], .)

Having obtained these pairings we shuffle them, that is we randomize the order in which the training instances are presented to the model during training for each epoch. This prevents the model from learning unintended patterns from the order of the data. Note that this approach to constructing the data set and the masking of future words of a text sequence is due to our specific task. In other settings, such as neural sequence-to-sequence translation, it is legitimate to use all tokens of a given sequence in the source language to produce the token of the next sequence that corresponds to the translation in the target sentence [SVL14].

2.1.3 *Neural Networks as Statistical Learners*

In 1999, Yoshua and Samy Bengio [BB99] introduced the idea of using a multi-layer neural network as a statistical language model. If f is the function the neural network computes to model p, $n-1$ the context length, $(w_{t-n+1}, \cdots, w_t)$ the sequence of context tokens and w_t the current token to predict we have

$$f(w_t, \cdots, w_{t-n+1}) = p(w_t | w_1, \cdots, w_{t-1}) \, .$$

In a follow up study in 2003, Yoshua Bengio [Ben+03] extended this set up by decomposing the function $f = g \circ \texttt{Embed}$:

1. Every token $w_i \in V$ (represented as an index or as a one-hot encoded vector) is mapped by a linear transformation $\texttt{Embed}$ to a real-valued vector $\texttt{Embed}(w_i) \in \mathbb{R}^e$. This vector represents the distributed feature vector for the token w_i.

2. g is defined as a function that maps an input sequence of feature vectors to a probability distribution over word sequences. Hence, g outputs a vector whose i-th element represents the probability for predicting the token with index i, denoted $p(w_t = i | w_1, \cdots, w_{t-1})$. In sum:

$$f(i, w_{t-1}, \cdots, w_{t-n+1}) = g(i, \texttt{Embed}(w_{t-1}), \cdots, \texttt{Embed}(w_{t-n+1})) \, .$$

Subsection 2.1.3.1 and Subsection 2.1.3.2 unpack the function $\texttt{Embed}$ and g, respectively. In parallel, Figure 2.1 visualizes how the neural network unfolds those functions.

2.1.3.1 *Word Embeddings*

The linear transformation function $\texttt{Embed}$ is identified with the $e \times |V|$ connection matrix W_e referred to as the *embedding matrix*. If a token is represented as an index j we index into the embedding matrix W_e retrieving a column $W_e[:, j]$. If the token is represented as a one-hot encoded vector $\vec{v}_j$,

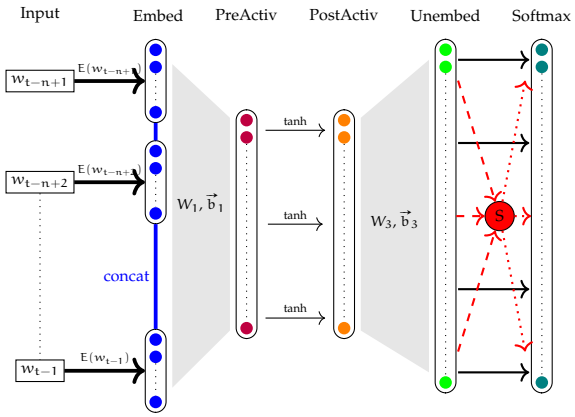

Figure 2.1: This figure illustrates the neural network employed by Bengio [Ben+03]. The diagram displays the parallel and iterative treatment of the context vectors through an embedding, denoted E, a linear and and activation layer followed by another linear unembedding layer. Finally the softmax function computes the probabilities. Notably, the red node, denoted S, computes the sum of the denominator necessary for the normalization step in the softmax function. The red node immediately visualizes a potential bottleneck that the softmax introduces in a neural network.

we similarly have $W_e[:, j] = W_e \vec{v}_j$. Hence, both approaches are equivalent to retrieve the *embedding vector* for the j-th token. Note that the embedding reduces the huge dimensionality of $\vec{v}_j$ significantly which can be interpreted as a form of *compression* of tokens. Moreover, we use the *same* embedding matrix for every vector of the context window referred to as *weight sharing*. It is crucial to understand that this practice leverages the underlying structure of our problem. We consider $W_e[:, j]$ as the distributed feature vector enriched with semantic meaning determined by the language we use. Specifically, we want that the same token in the context window at different places has the exact same representation because it represents a fixed semantic meaning.

Remark 5 (Human-Interpretable Embeddings) Learning the semantic structure of text and compression this text are known to be deeply related to each other: Learning procedures perform compression, and compression is an evidence of and is useful in learning [MY16]. As we noted, the neural network compresses the sparse high dimensional token vectors into dense low-dimensional embeddings. As the neural network improves in its predictions during training it concurrently learns to exploit the meaningful semantic and syntactic structure of the language by moving distributed feature vectors in direct neighborhood when they are structurally related. These language derived embeddings when projected to a lower dimensional space are human-interpretable [Şen+18]! In fact, this phenomenon is so wide-spread that one can easily replicate it. Figure 2.2 displays the embeddings of word-tokens that I obtained training a neural network (Experiment 1).

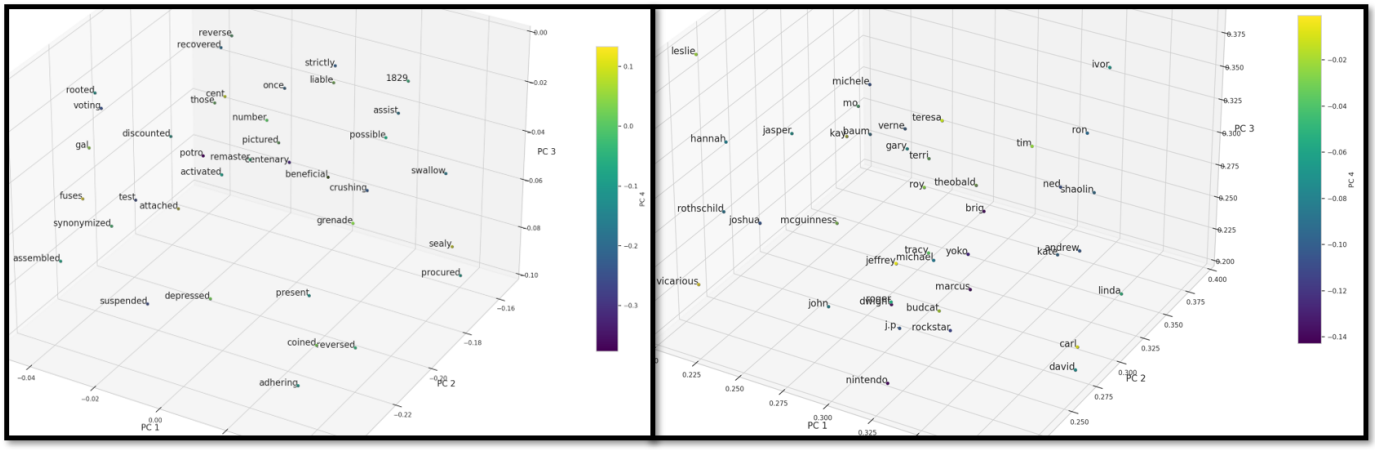

Figure 2.2: Each subplot displays a different region of the embedding space post principal-component reduction to a four-dimensional space. In both subplots, each point represents an embedded vector from the vocabulary. Whereas the subplot on the right highlights a region inhabited predominantly names, the left region hosts clusters of verbs and past-participles mainly associated with manipulating objects.

In this way we retrieve all the $n-1$ feature vectors corresponding to the $n-1$ words in the context window. These vectors are concatenated

$$\vec{x}_0 \leftarrow \text{Embed}(w_{t-n+1}) \circ \cdots \circ \text{Embed}(w_{t-2}) \circ \text{Embed}(w_{t-1}) \,,$$

and passed forward to the function g.

2.1.3.2 *Hidden States and Output Layer*

The function g receives the concatenated output $\vec{x}_0$ as its input and computes the vector $\vec{s} \in \mathbb{R}^{|V|}$ where $\vec{s}[i]$ estimates the probability $p(w_t = i|w_1, \cdots w_{ti-1})$. The first step is to pass $\vec{x}_0$ through two hidden layers L^1 and L^2 computing the pre- and postactivation respectively: Let p_1 be the number of hidden units of hidden layer L^1 and $\vec{d} \in \mathbb{R}^{p_1}$ its bias. We define the hidden layer weight matrix W_1 to be of size $p_1 \times (n-1)e$ and forward propagate $\vec{x}_0$:

$$\vec{h}_1 \leftarrow W_1 \vec{x}_0 + \vec{b}_1 \text{ forward propagation for Layer } L^1 \,,$$
$$\vec{h}_2 \leftarrow \tanh(\vec{h}_1) \text{ forward propagation for Layer } L^2 \,.$$

Afterwards we forward propagate through a linear *unembedding* layer L^3 with a $|V| \times p_1$ matrix W_3 with a bias vector $b_3 \in \mathbb{R}^{|V|}$:

$$\vec{h}_3 = W_3 \vec{h}_2 + \vec{b}_3$$

The purpose of the unembedding layer is to transform the dimension of the previous hidden layer to the dimension of $|V|$ and to assign each of the $|V|$ tokens an *unnormalized log-probability*. The entries of $\vec{h}_3$ are often referred to as *logits*. A high value $\vec{h}_3[j]$ indicates a high probability for the network

to predict the corresponding token. We note that log-probabilities can be negative and often logits are more open to interpretation than probabilities because their value is unaffected by the log-probabilities of other tokens.

However, what we really want our neural network to do is to output a probability vector. We already know how to do this: We map the log-probabilities to probabilities by applying the vector-to-vector softmax function s:

$$p(w_t|w_{t-1}, \cdots, w_{t-n+1}) = s(\vec{h}_2) ,$$

where $s(\vec{h}_2)[j] = s_j$ corresponds to the probability that the neural network assigns to the j-th token.

2.1.3.3 *Complexity Analysis of Learnable Parameters*

In order to show that the neural network represents an efficient method to fight the curse of dimensionality, we investigate the number of parameters the model has to learn [Ben+03]. The set of parameters, Θ, consists of the biases $\vec{b}_1$ with p_1 elements, $\vec{b}_3$ with $|V|$ elements, the matrices W_3 with $p_1 \cdot |V|$, W_1 with $p_1 \cdot (n-1) \cdot e$ elements, and W_e with $e \cdot |V|$ entries. To make it absolutely clear, the set of learnable parameters Θ can be written out as:

$$\Theta = [\vec{b}_1, \vec{b}_3, W_e, W_1, W_3] .$$

In sum, the number of parameters is

$$|V|(1 + ne + p_1) + p_1(1 + (n-1)e) = \mathcal{O}(|V|(ne + p_1) ,$$

where $\mathcal{O}$ denotes the *Landau* symbol. The important point to note here is that the number of parameters only scales *linearly* with the size of the vocabulary $|V|$ and context length $n-1$ thus overcoming the curse of dimensionality. Even until recently, main stream media refutes that large language models are capable of meaningful learning and attribute the performance of large language model largely to simple pattern matching and recalling probabilities [Hen22]. However, our complexity analysis provides us with a strong argument to refute such a claim: A large language model whose parameters only grow linearly with the size of the context length and size of the vocabulary simply does not have the space to memorize and store all conditional probabilities because we know from the curse of dimensionality that this requires exponentially growing space.

2.1.3.4 *Learning through Gradient Descent*

Training is achieved by maximizing the log-likelihood of the training data. That is, given a training instance consisting of a context and a target word with index j a gradient descent update is performed for all $\theta \in \Theta$:

$$\forall \theta \in \Theta : \theta \leftarrow \theta + \alpha \frac{\partial \log p(w_t = j|w_{t-1}, \cdots, w_{t-n+1})}{\partial \theta} . \tag{2.2}$$

2.1.4 *Evaluating Performance*

This subsection presents two metrics for assessing the performance of a large language model after training. In this vein, cross-validation serves as a methodology for obtaining an *unbiased* estimation in the sense that our metric measures the true performance in an expected sense ([Bis95], Chapter 9). In Subsection 2.2.1.1 we make this requirement mathematically precise. To implement cross-validation, the data sample $\mathbb{X}$ is partitioned into three sets:

- **Training Set ($\mathbb{X}^{train}$)**: Dedicated to the primary training process.
- **Validation Set ($\mathbb{X}^{valid}$)**: Allocated for hyperparameter adjustment and early termination of training.
- **Test Set ($\mathbb{X}^{test}$)**: Used exclusively for the model's final evaluation.

Critically, the separation of data sets allows for an unbiased tuning of hyperparameters and an unbiased approximation of the true generalization error ([Bis95], Chapter 9).

2.1.4.1 *Empirical Generalization Error*

The *empirical generalization error*, denoted $\hat{C}$, serves as an estimator of the true generalization error and is expressed as:

$$\hat{C} = \text{mean}_{(\vec{x},\vec{y})\in\mathbb{X}^{test}} \left[\mathcal{L}(f_{\eta^+}(\vec{x}), \vec{y})\right] \ ,$$

where f_{η^+} represents the best-fit model for the optimized choice of hyperparameters η^+.

2.1.4.2 *Perplexity*

Another common metric is referred to as *perplexity* a measure borrowed from information theory [Jel+77]. In this realm, perplexity is the exponentiation of the average negative log probability per canonical prediction unit [Ben+03]. In language prediction specifically, we compute the *normalized* perplexity, denoted $\mathcal{P}$ [Agg20]: Given a word sequence, $\sigma = (w_1, \cdots w_t)$ and a language model, p, its normalized perplexity is:

$$\mathcal{P}(w_1, \cdots w_t) = \exp\left(-\frac{1}{t}\sum_{i=1}^{t} \ln(p(w_i|w_{i-1}, \ldots, w_{i-n+1}))\right) .$$

If we want to know the perplexity for the whole corpus containing m sequences $(\sigma_1, \cdots, \sigma_m)$ and t words all together, we have to find out how well the model can predict all the sentences together. Assuming that the sentences are shuffled and drawn independently from each other the perplexity of the corpus is defined as

$$\mathcal{P}(\sigma_1, \cdots \sigma_m) = \exp(\left(-\frac{1}{t}\sum_{i=1}^{m} \ln(p(\sigma_i))\right) ,$$

where I have corrected a mistake in the definition in [Agg20]. The ultimate goal is to minimize $\mathcal{P}$ which corresponds to enhancing the model's predictive capability.

2.1.4.3 *Pushing Performance*

The first convincing and successful attempt to train a number of large neural networks to predict the next word was performed using the *Brown University Standard Corpus of Present-Day American English* corpus [Ben+03]. This corpus is a collection of text samples of American English, the first major structured corpus of varied genres that contained over 1 million words. The best model reported by Bengio [Ben+03] consisted of two hidden layers with an embedding dimension of 60 and a hidden dimension of 50 and achieved a then state-of-the-art perplexity of 268. I was able to dramatically improve on this metric for three reasons:

1. More and better data: I have used the WikiText-2 language modeling data set which is a collection of over 100 million words extracted from the set of verified articles on Wikipedia.

2. Specialized Hardware: Whilst Bengio et al. [Ben+03] used CPUs I use a GPU (Geforce GTX 4080) allowing for massive parallelization.

3. Research Breakthroughs: In Chapter 3 we will study a number of algorithmic advancements that led to substantially better performance of neural networks.

By exploiting these three factors, I was able to achieve a validation perplexity of ~ 3.6 (Experiment 2), which serves as a foretaste for chapter 3.

2.2 THEORY FOR INDUCTIVE REASONING

This section attends to the question, *why do neural networks work at all?*, alluded to in the introduction. Large language models should be seen from the perspective of *inductive probability*, a source of knowledge which attempts to give the probability of future events based on past events [Sol64]. Inductive probability is the basis for inductive reasoning, and allows to establish new facts from data. This section discusses two important frameworks for inductive reasoning: *maximum likelihood estimation* (Subsection 2.2.1) and *minimum description length* (Subsection 2.2.3). The most commonly encountered way of thinking in machine learning text books is the maximum likelihood point of views even though we will see that the underlying assumptions of this frameworks are not met [GBC18] [Agg23]. Furthermore, the maximum likelihood approach fails due to the phenomenon of *overfitting* (Subsection 2.2.2). In contrast, we show that the minimum description length principle is a potent framework to explain the success of deep learning.

2.2.1 *Maximum Likelihood Estimation*

The neural network training procedure minimizes the cross-entropy between the ground truth distribution q and the neural network probability distribution p. From a general perspective, we pick parameters that maximize the probability of the observed ground truth tokens in a given training set. This is exactly the strategy employed by *maximum-likelihood estimation* [Ros18]. The implicit assumption is that the empirical data observed in the training set is *typical* for the unknown probabilistic data source q. Thus, being able to perform well on the training set is assumed to typically lead to a good performance on any data set drawn from q [GBC18]. Definition 4 makes this more precise:

Definition 4 Let $\mathbb{X}^{train} = \{(\vec{x}_1, \vec{y}_1), \ldots, (\vec{x}_m, \vec{y}_m)\}$ be an independently drawn sample of the true but unknown distribution q. We define $p_{\Theta \in \hat{\Theta}}(\vec{x}_i)$ to be the parametric family of probability distributions indexed by the learnable parameters Θ from the parameter space $\bar{\Theta}$. Then, the maximum-likelihood estimator $\Theta_{ML} : \mathbb{X}^{train} \to \bar{\Theta}$ picks the parameters Θ_{ML} that minimize the neural network's loss, denoted $\mathcal{L}$:

$$\Theta_{ML} := \arg\min_{\Theta \in \bar{\Theta}} \mathcal{L}(p_\Theta(\mathbb{X}^{train}, q(\mathbb{X}^{train})) \tag{2.3}$$

$$= \arg\min_{\Theta \in \bar{\Theta}} \prod_{i=1}^{m} \mathcal{L}(p_\Theta(\vec{x}_i, q(\vec{y}_i)) \; . \tag{2.4}$$

However, Equation 2.4 is numerically unstable. Hence, we use the equivalent log-likelihood optimization problem:

$$\Theta_{ML} = \arg\min_{\Theta \in \hat{\Theta}} \sum_{i=1}^{m} \log \mathcal{L}(p(\vec{x}_i), q(\vec{y}_i)) \; .$$

The estimator Θ_{ML} may be viewed as an *inductive learning algorithm* where learning is mapping the input data to a choice of parameters. Notably, to solve this optimization problem we employ stochastic gradient descent, a method that does not necessarily find the global minimum of the loss. Hence, we realize that we may end up with a *local* maximum likelihood solution.

2.2.1.1 *Generalization*

The reader may object that the goal of neural network learning is not to maximize the probability of the training data. What we really want is to learn an algorithm that performs well on unseen data. To make this more precise we define the *generalization error*:

Definition 5 (Generalization Error) Let q be the true but unknown distribution of the data that the neural network wants to approximate be modeling

a function f. Then, given a loss function, denoted $\mathcal{L}$, the generalization error is defined by

$$C := \mathbb{E}_q \left[\mathcal{L}(f(\vec{x}), \vec{y}) \right] .$$

The empirical loss on the training set is a surrogate for the true generalization error C but have no direct access to. So we should convince ourselves that as long as the amount independently sampled of training data originating from q increases, the training algorithm causes the empirical loss on the training set and the generalization error to converge [Ben12]. Since we use cross-entropy as the loss function, $\mathcal{L}$ we call our learning algorithm *consistent* if q converges to p in the sense that $\mathcal{L}(p, q) \to 0$ with q-probability 1. Thus, given enough data, the learning algorithm should learn a good approximation of q with high probability [Grü05]. In Example 1 we show that our maximum likelihood algorithm works in the case of *online learning*. Bishop ([Bis95], Chapter 9) showcases a more general treatment of this problem.

Example 1 (Online Learning) In this example we make the simplified assumptions that the probability mass function q, the loss function $\mathcal{L}$, and the neural network function f are differentiable. Furthermore, we assume that the neural network has access to an unlimited stream of training tuples $\{(\vec{x}_1, \vec{y}_1), (\vec{x}_2, \vec{y}_2), \cdots\}$. Receiving a random tuple $(\vec{x}, \vec{y}) \in \mathbb{R}^{in} \times \mathbb{R}^{out}$ of the input-label space the training algorithm computes for every parameter $\theta \in \Theta$ the stochastic gradient:

$$\hat{g}(\vec{z}) := \frac{\partial \mathcal{L}(f(\vec{x}, \vec{y}))}{\partial \theta} .$$

Assuming the simplified case of independently and identically distributed examples we have by definition:

$$C = \mathbb{E}_q \left[\mathcal{L}(f(\vec{x}), \vec{y}) \right] = \int_{\mathbb{R}^{in} \times \mathbb{R}^{out}} \mathcal{L}(f(\vec{x}), \vec{y}) q(\vec{x}, \vec{y}) d(\vec{x}, \vec{y}) .$$

Since the functions in the integral are continuously differentiable we have:

$$\frac{\partial C}{\partial \theta} = \frac{\partial}{\partial \theta} \int \mathcal{L}(\vec{x}, \vec{y}) q(\vec{x}, \vec{y}) d(\vec{x}, \vec{y}) = \int \frac{\partial \mathcal{L}(\vec{x}, \vec{y})}{\partial \theta} q(\vec{x}, \vec{y}) d(\vec{x}, \vec{y}) = \mathbb{E}[\hat{g}] .$$

This shows that the difference between the expected value of the estimator $\hat{g}$ and the true value of the generalization error gradient $\frac{\partial C}{\partial \theta}$ is 0. In conclusion, reducing the training loss will result in a reduction of the error on new and unseen data [BB12].Given an endless stream of training data we can optimize our neural network and decrease the generalization error [BB12].

2.2.1.2 *Problems with Maximum Likelihood*

Unfortunately, there are two fundamental objections to maximum likelihood:

1. In real settings, we do not have infinite data. In consequence, when we lack enough data maximum likelihood is prone to overfitting (see

Subsection 2.2.2). Briefly explained, overfitting occurs when the model matches the random noise and not the pattern in the data [Agg23].

2. Moreover, maximum likelihood can only approximate the true distribution when q lies in the parametric family defined by $p_\Theta(\cdot)$ [Grü05]. For instance, when modeling text to speech on internet data it seems obvious that we cannot assume q to be a neural network generating all text of the internet.

2.2.2 *Overfitting*

Typically, overfitting is observed when the network achieves a low loss on the training data whereas the model reveals a low performance on unseen data. In simple neural network architectures this problem can occur when the number of number of parameters of the model exceeds the training examples. We illustrate this concept in Example 2 adapted from [Agg23].

Example 2 (Example of Overfitting) Table 2.2 contains four training instances $(\vec{x}_i, \vec{y}_i) \in \mathbb{R}^5 \times \mathbb{R}$. To model this data we apply a neural network consisting of a single-layer neural network with the identity function as the activation:

$$\hat{\vec{y}} = \vec{w}^\intercal \vec{x} \, .$$

Suppose that the true relationship between the inputs and outputs is known that is that the target is solely dependent on the first variable x_1 of the input vector. Hence, the network is supposed to learn the parameter vector $\vec{w} = (10, 0, 0, 0, 0)^\intercal$. However, since the number of training points is lower than the number of parameters there are an infinite number of solutions with 0 loss. For example, the parameter vector $\vec{w} = (0, 10, 20, 30, 40)^\intercal$ would achieve 0 loss. However, a model with this $\vec{w}$ will perform poorly on unseen test data because the learned parameters ignore the first variable which is the true predictor of the target values. Instead, random noise in the data is learned.

	Training Examples					Target
	x_1	x_2	x_3	x_4	x_5	y
$\vec{x}_1$:	1	1	0	0	0	10
$\vec{x}_2$:	2	0	1	0	0	20
$\vec{x}_3$:	3	0	0	1	0	30
$\vec{x}_4$:	4	0	0	0	1	40

Table 2.2: Training Set to Illustrate Overfitting

2.2.2.1 *Tackling Overfitting: Weight Decay*

Subsection 2.2.2 leads to the insight that it might be advantageous to ensure that there is less information in the parameters than there is in the training instances. There are numerous ways to implement this idea such as limiting the precision of the parameters, weight sharing based on the natural symmetries of the task, or restricting the number of connections in the network [HVC93]. Another intuitive idea to fight overfitting is to directly constrain the model to a sparse set of parameters. Suppose in Example 2 we constrain the parameter vector $\vec{w}$ to have only one non-zero component. Then, the only zero loss solution would be the correct solution $(10, 0, 0, 0, 0)^\intercal$. In practice, a soft constraint is used. That is the model is encouraged to learn small absolute values for all parameters $\theta \in \Theta$. This is achieved by adding a weight penalty $R(\Theta)$ to the loss function $\mathcal{L}$ where $R(\Theta)$ is defined as

$$R(\Theta) = \lambda \sum_{\theta \in \Theta} |\theta|^p \ .$$

The regularization parameter $\lambda \in \mathbb{R}$ constitutes another hyperparameter regulating the softness of the weight penalty. Setting $p := 2$ leads to *Thikinov regularization*. Setting $p := 1$ leads to L1 regularization. Thikinov regularization punishes large values more strongly than L1 regularization and often outperforms L1 regularization [Agg23]. Regularization necessitates a modification of the parameter updates for gradient descent: First every parameter is multiplied with the weight decay factor $(1 - \alpha \cdot \lambda)$ and *then* the stochastic gradient-descent update is performed:

$$\begin{aligned} &\text{Step 1: } \forall \theta \in \Theta : \theta \leftarrow \theta(1 - \lambda\alpha) \\ &\text{Step 2: } \forall \theta \in \Theta : \theta \leftarrow \theta - \alpha \frac{\partial \mathcal{L}}{\partial \theta} \ . \end{aligned}$$

In practice, regularization may decrease the accuracy of the model on the training data however improves the accuracy on the test data set [Agg23]. Using a model with economy in parameters decreases the expressivity of the model. In fact, when the data patterns are too complex the model is not capable to model it. This is exactly the opposite situation of overfitting and is called *underfitting*. This introduces a trade-off where we have to decide when the extra complexity in the model is worth the improvement in the data-fit.

2.2.3 *Minimum Description Length*

In Subsection 2.2.2 we observed that relying solely on maximum likelihood results in overfitting. To mitigate this, we incorporated a weight decay term into the loss function. On the upside, this term introduces a beneficial trade-off between model simplicity and training error, effectively reducing overfitting. However, this modification also necessitates the development of a new theoretical framework to give us a basic understanding of what learning in

the context of neural networks means. In essence, we are in search of a theory of inductive inference that aligns with the following requirements for model selection:

1. theory should naturally choose a model that trades off goodness-of-fit with model complexity.
2. In most deep learning settings it is not reasonable to assume that there is an underlying ground truth distribution that we can learn. We want a theory that also works in this situation.
3. theory should select a model that performs best on unseen data.

The core idea to address these challenges lies in a well-established fact [Ris86]: mathematically, predicting a sequence of binary numbers given an observed sequence of these numbers is the same problem as compressing that sequence. Importantly, this compression must be done by choosing a coding scheme *before* seeing the data. To make this more tangible, we reproduce an important result found by Jorma Rissanen [Ris86]: *The greatest lower bound for encoding binary digits constitutes also the greatest lower bound for the prediction errors that result when the same data is predicted*. This insight helps us to redefine our goal: We're in search of a theory that identifies the model that compresses the data the most because we assume that it is also the best model to predict unseen data [Grü05].

2.2.3.1 *MDL's Philosophy*

The minimum description length (MDL) principle provides a universal approach to inductive inference, drawing a direct parallel between learning and data compression. When selecting from a range of parameterized models, MDL prioritizes models that strike a balance between their inherent complexity and predictive accuracy.

The complexity of a parameterized model is determined by the number of distinguishable probability distributions the model contains [Grü05]. Intuitively, if a model contains a high number of distributions it can fit more data patterns than a model of lower complexity. Moreover, the more complex a model is, the bigger is its *description length*, that is the number of bits needed to unambiguously describe the model. The accuracy of the model is determined by its ability to fit the observed data. Analogously, the description length of the data decreases as the model is more accurate since the model only needs to describe the difference between its prediction of the data and the true value of the data. The MDL principle selects the model that minimizes the data's *stochastic complexity*. The stochastic complexity of the data is the cumulative code-length of both the model's accuracy and its inherent complexity [Grü05].

MDL is agnostic to the potential existence of an ounderlying ground truth distribution: In this sense uses any regularity in the data to describe it using fewer symbols than the number of symbols needed to describe the data literally. According to Rissanen [Ris86], the originator of this princple, in MDL

understanding the observed data means the ability to remove redundancies in the data and to discover regular statistical features. If the shortest description of the data is found in terms of the models there is nothing further we can learn about the data.

2.2.3.2 *Translating Supervised Learning to Compression*

Shortly after its formalization, Hinton and Drew Van Camp [HVC93] discovered the connections between MDL and deep learning and used them to improve neural network performance. We describe how a supervised machine learning setting is translated to a compression task [BO18]:

Consider a classification scenario with a data set, denoted $\mathbb{X}^{train}$, consisting of pairs $(\vec{x}_1, \vec{y}_1), \cdots, (\vec{x}_n, \vec{y}_n)$. For concreteness, let's take $\mathbb{X}^{train}$ to be the CIFAR-10 dataset, a widely-used machine learning benchmark comprising $50,000$ training images of size 32×32 distributed across ten distinct classes such as airplanes, cars, and birds. In the compression scenario, Alice possesses the full dataset $\mathbb{X}$, while Bob only has access to the images $\{\overrightarrow{x_1}, \cdots, \overrightarrow{x_n}\}$. The task for Alice is to efficiently convey the labels to Bob.

First, we establish a baseline for compression using a uniform encoding scheme, that is the labels of the data set are directly encoded without leveraging any image-based models. The computational cost in the case of CIFAR10, equates to $50,000 \times \log_2 10 \approx 166$ kbits.

2.2.3.3 *Coding Schemes in MDL*

One obvious way to instantiate the MDL principle is the called *two-part code-length* where use standard float32 binary encoding to send the trained model's parameters and to send the prediction errors in parallel. For a trained model with 1 Million parameters, this two-part code encoding will amount to 32 Mbits [BO18]. Indeed, one of the great mysteries of modern deep learning is the fact that in practical settings models with many millions of parameters can potentially fit the fit training data perfectly with 0 error and still, they perform very well on future test sets [Zho+18]. Moreover, given the superior performance of huge neural networks on unseen data, it seems at first sight that the best model for compression is not the best model that predicts unseen data. In the following, we show that this first intuition is incorrect and shed some light in this mystery: namely, deep neural networks can be described with far fewer parameters and consequently the number of parameters is not an obstacle to compression.

There are several alternative ways to implement the MDL principle such as *variational techniques* [HVC93] and the *Bayesian universal approach* [Grü05]. Theoretically, their respective code-lengths converge under strong assumptions. However, in deep learning, these assumptions are rarely met requiring a clever approach [Grü05].

In this line, Blier and Ollivier [BO18] recently showed that *prequential coding* yields much better code-lengths than the previous methods. Furthermore, they showed that the model's ability to compress the data correlates

with its performance on unseen data. The idea behind prequential code is the following: First, Alice sends Bob the description of a network architecture, a description of the optimization algorithm, a seed for a random sequence generator for the (stochastic) optimization algorithm, and the first 100 labels of the dataset compressed through uniform encoding. Then, Alice and Bob both train the network with these 100 labels. Alice can now use this trained network as a model to encode the 100 next labels. Now they both have 200 labels and can train a better model, which will be used for the next 100 labels. As more labels are transmitted, the model becomes more accurate, and sending more labels is less expensive. Intuitively, the compression scheme in prequential coding leverages the generalization ability of deep learning networks, even with very limited data sets. Interestingly, even though the model parameters are never encoded explicitly Bob computes the same model as Alice and possesses all the labels at the end of the procedure [BO18].

Using this coding scheme Blier and Ollivier [BO18] achieved a codelength of 45.3 kbits on the CIFAR10 dataset and a 93 percent accuracy on the test set. The performance of the prequential approach indicates that deep learning models can represent the data together with the model in fewer bits than uniform encoding and is consistent with the notion that neural networks can be seen as statistical models and *also* as compressors [BO18]. In conclusion, where maximum likelihood falters due to overfitting, MDL inherently favors a balance between model simplicity and fit. Moreover, MDL provides a new perspective on the initially described mystery that contradicts almost any statistical theory: Not the ratio of of parameters count to the number of training examples is important to estimate the generalization capability of a neural network but its ability to compress. Here, compression is measured by the stochastic complexity of the data constituted by the number of bits needed to *describe* the model and the data given the model [GR19].

2.3 REPRESENTATIONAL POWER OF NEURAL NETWORKS

Having established a theoretical framework for neural network learning, the next question is: How powerful are these networks? In short: Very! In Subsection 2.3.1, we formally prove Theorem 1 that neural networks with only two hidden layers are already *universal function approximators* [SSS14]. In Subsection 2.3.2.1, we see that we cannot simplify neural networks by excluding the computationally intensive activation functions, as they are essential to their expressiveness [Agg23]. Even though Theorem 1 is encouraging it does not directly translate into practical usefulness because the two layer neural network employed in the proof of Theorem 1 grows exponentially in the input size. In this vein, the last Subsection 2.3.2 offers a recourse to this dilemma: Increasing the depth of neural networks makes the network more efficient in an exponential manner allowing for greater representational power with fewer hidden units.

2.3.1 *Going Wide*

Theorem 1 posits that a neural networks with only two hidden layers can approximate every Boolean function $g : \{\pm 1\}^n \to \{\pm 1\}$ where ± 1 denote the binary Boolean values. First, we illustrate why this is of fundamental importance: Let us assume our computer represents real numbers using 32 bits. Whenever we compute any function $f : \mathbb{R}^n \to \mathbb{R}$ on that 32-bit machine, in fact we calculate a Boolean function $g : \{\pm 1\}^{n \cdot 32} \to \{\pm 1\}^{32}$. Thus, proving that a neural network can approximate every Boolean function is equivalent to showing that a neural network can implement all functions our computer can calculate [SSS14].

Before we prove Theorem 1 we need to (re-)introduce some notation. Recall the definition of a neural network architecture: $\mathcal{A} = (V, E, \Phi)$ from Subsection 1.1.4. We set $\Phi := \mathtt{sign}$ and define a hypothesis class, denoted $\mathcal{H}_{V,E,\mathtt{sign}}$:

$$\mathcal{H}_{V,E,\mathtt{sign}} := \{f_{\mathcal{A},\Theta} | \Theta \in \hat{\Theta}\}\,, \tag{2.5}$$

where the neural network function $f_{\mathcal{A},\Theta}$ is defined by the concrete neural network architecture and the chosen set of weights Θ in the weight space $\hat{\Theta}$. Notably, the hypothesis class $\mathcal{H}_{V,E,\mathtt{sign}}$ contains all functions a particular neural network architecture identified by $\mathcal{A}$ can implement [SSS14]. Now, we are ready to formulate Theorem 1:

Theorem 1 *For every $n \in \mathbb{N}$, there exists a neural network with two hidden layers (one is the output layer), such that $\mathcal{H}_{V,E,\mathtt{sign}}$ contains all Boolean functions from $\{\pm 1\}^n$ to $\{\pm 1\}$.*

Proof

We fix the number of neurons in the input layer $|\, V^0 \,| := n + 1$ where the last neuron contains the bias neuron. Furthermore, set $|\, V^1 \,| = 2^n + 1$ and for the output layer $|\, V^2 \,| = 1$. Let $B : \{\pm 1\}^n \to \{1\}$ be an arbitrary but fixed Boolean function. The proof is complete when we show that there exists a choice of weights so that the network will implement B. Let $\vec{x}_1, \cdots, \vec{x}_k$ be all n-dimensional input vectors with $B(\vec{x}_i) = 1$. Note that $k \leqslant 2^n$. For every very $\vec{y} \in \{\pm 1\}^n$ we have the following two cases: If $\vec{y} \neq \vec{x}_i$ then the dot product between $\vec{x}_i$ and $\vec{y}$ results in: $\vec{x}_i \cdot \vec{y} \leqslant n - 2$ and if $\vec{y} = \vec{x}_i$ we have $\vec{y} \cdot \vec{x}_i = n$. Thus, the function $g_i(\vec{y})$ defined by

$$g_i(\vec{y}) = \mathtt{sign}((\vec{y} \cdot \vec{x}_i) - n + 1) \tag{2.6}$$

equals 1 if and only if $\vec{y} = \vec{x}_i$. Hence, we can choose the weights between the input layer V^0 and the hidden layer V^1 so that for every $1 \leqslant i \leqslant k$, the

i-th neuron in the hidden layer implements the function $g_i(\vec{y})$. Since the function B is the disjunction of the functions $g_i(\vec{y})$, f can be written as

$$B(\vec{y}) = sign\left(\sum_{i=1}^{k} = g(\vec{y}) + k - 1\right) \tag{2.7}$$

which concludes the proof. ∎

Theorem 1 shows that neural networks can implement any Boolean function under the premise that we allow the size of the hidden layer to grow exponentially with the input. Shai, and Shai [SSS14] show that this property is not an artifact of the proof or the specific activation function chosen. In other words, a neural network architecture where V is of polynomial size that can implement every Boolean function does not exist [SSS14].

2.3.2 *Going Deep*

While Theorem 1 establishes that a large enough two-layer neural network can approximate any function, empirical evidence suggests that deeper architectures often outperform shallow ones. A seminal turning point demonstrating the power of deep architectures was the introduction of AlexNet [KSH12], an eight-layer deep neural network. Introduced by Krizhevsky et al. in 2012, AlexNet achieved a landmark victory in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), which evaluates algorithms for object detection and image classification at large scale outperforming all other algorithms by a significant margin [KSH12]. This groundbreaking performance lead to a paradigm shift towards deeper architectures. In order to understand what makes deep networks so effective we need to examine the crucial role of the compositional power of nonlinear activation functions.

2.3.2.1 *Importance of Nonlinearity*

As a first step I proof Lemma 1 demonstrating that nonlinear activation functions are essential parts of any neural network architecture:

Lemma 1 (Purely Linear Neural Networks are not Powerful) *Let $W_1, W_2, \cdots, W_n$ be linear transformations. Then, the composition of these linear transformations, denoted $W(\cdot)$, is itself a linear transformation.*

Proof I proceed by induction.
Base Case: First, we consider the base case where $n = 1$, that is, there is only one linear transformation $W = W_1$. By definition, W_1 is a linear transformation, satisfying the properties of additivity and homogeneity:

$$\begin{aligned} W(x+y) = W_1(x+y) = W_1(x) + W_1(y) = W(x) + W(y), \text{ and} \\ W(cx) = W_1(cx) = cW_1(x) = cW(x). \end{aligned}$$

Inductive Hypothesis: Now, we assume that the proposition holds true for a $n \geqslant 1$, $W(\cdot) = W_n \circ W_{n-1} \circ \cdots \circ W_1(\cdot)$, that is:

$$W(x+y) = W(x) + W(y), \text{ and}$$
$$W(cx) = cW(x).$$

Inductive Step: Next, we consider the case where $W = W_{n+1} \circ W_n \circ \ldots \circ W_1$ is a composition of $n+1$ linear transformations. Utilizing the properties of the linear transformation W_{n+1}, leveraging the fact that $W = W_{n+1} \circ \hat{W}$ with $\hat{W} = W_n \circ W_{n-1} \circ \cdots \circ W_1$ and applying the inductive hypothesis on $\hat{W}$, we can write:

$$\begin{aligned}
W(x+y) &= W_{n+1}(\hat{W}(x+y)) = W_{n+1}(\hat{W}(x) + \hat{W}(y)) \text{ (by the inductive hypothesis)} \\
&= W_{n+1}(\hat{W}(x)) + W_{n+1}(\hat{W}(y)) \text{ (by the additivity of } W_{n+1}) \\
&= W(x) + W(y), \text{ and} \\
W(cx) &= W_{n+1}(\hat{W}(cx)) = W_{n+1}(c\hat{W}(x)) \text{ (by the inductive hypothesis)} \\
&= cW_{n+1}(\hat{W}(x)) \text{ (by the homogeneity of } W_{n+1}) \\
&= cW(x) \ .
\end{aligned}$$

Thus, by induction, I have proved that the composition of any number of linear transformations is itself a linear transformation. ∎

Corollary 1 *In a neural network containing only linear functions, increased depth does not result in increased complexity, as the entire network can be reduced to a single linear transformation.*

2.3.2.2 *Leveraging Repeated Regularities in the Data*

Having established the critical role of nonlinearity in neural networks, we investigate the revealing question how depth and nonlinearity combine to capture complex patterns in data. Albeit, due to its complexity, the deep learning machinery often resists conventional formal proofs. As a workaround, the field leans on the scientific method: hypotheses stemming from initial mathematical conjectures are tested through empirical studies. This subsection endeavors to present the general idea explaining the benefits of depth in neural network architectures whilst omitting mathematical details.

First, we observe that each hidden layer in a deep neural network can be viewed as a computational unit that transforms the data representation. This transformation is not arbitrary but is guided by the optimization process and the architecture's ultimate goal: For instance, Aggarwal [Agg23] suggests that when the output layer employs a linear separator for classification, the backpropagation algorithm sends loss signals that encourage hidden layers to transform the input data linearly separable through nonlinear transformations. This leads to an interesting assumption: depth allows the neural network to divide the labor by first successively creating representations of the data that are more informative for a predictive task. Afterward,

the final layer can leverage these representations for prediction. This form of division of layer is referred to as *hierarchical feature engineering*.

Next, we investigate the deep neural network's ability to exploit repeated structures in data more closely. In essence, neurons in later layers reuse computations performed by earlier layers, acting as building blocks to generate increasingly complex features. This computational reuse is not merely a repetition; it allows for the sophisticated interaction and combination of linear regions formed in earlier layers, enhancing the network's ability to model complex relationships in data.

In this vein, it is essential to understand the concept of a linear region [PMB13]. A linear region within a neural network can be described as a contiguous segment of the input space where the network's response is linear. For instance, for a given neuron with ReLU activation, the input space is divided into two linear regions: one where the neuron is active and one where it is inactive. As the input passes through successive layers, each layer's nonlinear transformations further divide the space into smaller regions. Critically, Pascanu et al. [PMB13] demonstrate that deep networks exhibit exponential growth in the number of distinct linear regions. This exponential increase in linear regions increases the representational capacity of deep networks, enabling them to model more complex and high-dimensional data.

Furthering this line of research, Montúfar et al. [Mon+14] showed that intermediate layers create what can be termed as 'neighborhoods', by mapping multiple linear regions to the same output regions. These 'neighborhoods' represent a more efficient representation, as they allow the network to reuse and recombine computations performed in these linear regions across different layers. This process of reusing and refining computations in 'neighborhoods' greatly contributes to the overall efficiency and depth-enhanced complexity of deep neural networks [Mon+14].

2.3.2.3 *Visualizing the Benefits of Increased Depth*

The following discussion aims to demonstrate the above theoretical discussion in a more tangible way. Given the function $g : \mathbb{R}^2 \to \mathbb{R}^2$ by

$$g(x_1, x_2) = (|x_1|, |x_2|)^T$$

the four quadrants of the 2 dimensional Cartesian plane are identified. The reader can verify that g corresponds to folding the space once along the x-axis and once along the y-axis. In Figure 2.3, a composition of space folding is depicted. Each hidden layer can be associated with such a folding. Naturally, the weight and bias parameters of a layer allow for a folding that does not preserve length and allows for shifts and different orientations than the coordinate axes. Figure 2.4 displays an example where the sizes and orientations of identified neighborhood regions differ [Mon+14]. The crucial point to understand is that the computations carried out in the lth layer on the set of activations from the previous layer of the neural network corresponds to

carrying out these computations in all regions of the *input space* that map to the same activations of the $(l-1)$th layer [Mon+14].

Again, we visualize this idea in Figure 2.3: Observe the computation done in the region S of the second layer marked by the orange lines. Note how this pattern is replicated in $S'1, S'2, S'3, S'4$ in all four quadrants $S1, S2, S3, S4$. These regions are identified as neighborhoods by the intermediate layers of the neural network, and thus all map to S. Furthermore, observe how the simple pattern in the second layer appears intricate and interesting, generating an overall complicated-looking function with very few parameters due to the power of function composition. Also, note how this example illustrates how a folding along two axes leads to an exponential identification of input regions.

In conclusion, the general principle of assembling complex features from basic features allows deep models to compute functions that react equally to complicated patterns of different inputs that are mapped to the same output [Mon+14]. Now, that we have explored the mathematical and empirical justifications for depth in neural networks we can ask: How does this depth manifest in learned features? In the next subsection, we use visualization techniques to get a closer look.

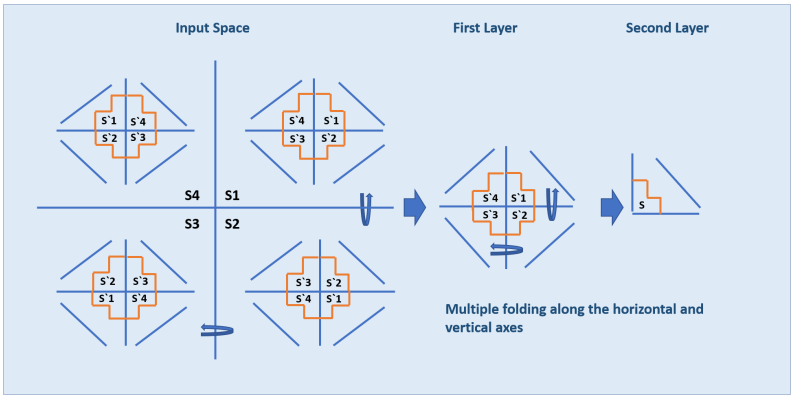

Figure 2.3: The 2 dimensional plane is folded two times along the x-axis and along the y-axis identifying 16 regions of the input space in the second layer. The simple pattern computed in the second layer is replicated in all neighborhoods whose outputs are mapped to the set of activations in the first layer that map to the region S in the second layer. Adapted from [Mon+14].

2.3.2.4 *Visualizing Hierarchical Feature Engineering*

In the context of image input, hierarchical feature engineering process becomes evident: The early layers of image classifier networks typically focus on extracting simple, generic features such as edges, lines, and hexagons. As we progress through the network, these primitive features serve as building blocks for the later layers, which then assemble them into more complex and semantically meaningful patterns relevant to the target class labels. To illustrate this hierarchical feature abstraction in a tangible way, techniques like

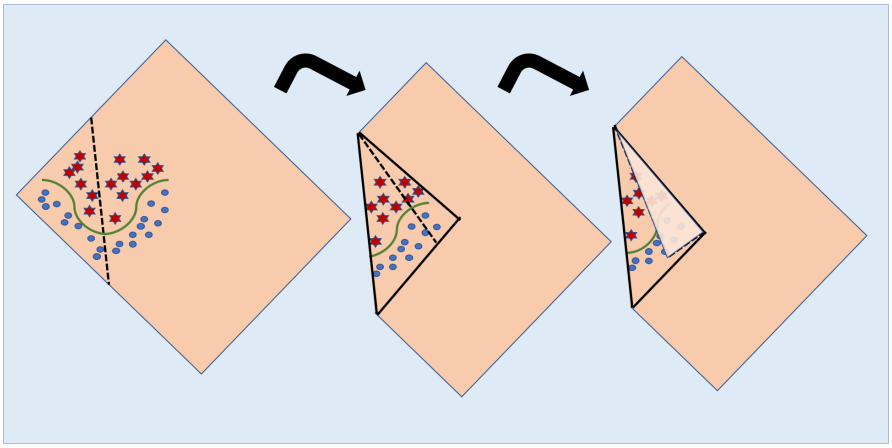

Figure 2.4: The two dimensional space is folded in a way that captures identified symmetries in the data that are not oriented along axes of the chosen coordinate system. Adapted from [Mon+14].

DeepDream created by Google engineer Alexander Mordvintsev [MOT15] can be employed.

This computer vision algorithm takes a trained neural network specialized in image classification and inverts its functionality [MV15]. Instead of identifying features in an input image, DeepDream tweaks an initial image filled with random noise to maximize the activation of a specific layer. This process is regulated by further constraints that make the resulting image statistically similar to natural images [OMS17]. The end result visualizes the features that a particular layer has been trained to recognize.

In this context, *GoogLeNet* [Sze+15] serves as an illustrative example. GoogLeNet is a deep neural network optimized for image classification tasks, comprised of 22 layers. Figure 2.5 presents the outcome of applying DeepDream to specific layers of this architecture. In the early layers, labeled *3a* and *3b*, the activations predominantly reflect basic geometric and textural primitives such as lines, edges, and textures. As one progresses to the intermediate layers, specifically *4a* and *4b*, the activations become noticeably more intricate. These layers respond to composite geometric arrangements, which can be conceptualized as higher-order combinations of the primitive features detected in the initial layers. Finally, the activations in the proximate output layers, denoted *5a* and *5b*, correspond to an array of abstract concepts.

To demonstrate the main insight that increased depth can lead to computationally simpler and yet more accurate models we compare GoogLeNet to the previously introduced AlexNet: despite having only eight layers, AlexNet comprises more than 60 million parameters. In contrast, GoogLeNet, with its 22 layers, contains only 6.5 million parameters, yet it outperforms AlexNet by a significant margin [Sze+15].

2.3.2.5 *Depth's Dilemma: Vanishing and Exploding Gradients*

Depth is undoubtedly a key enabler of the sophisticated capabilities of modern neural networks. Here, a natural question arises: if depth is advanta-

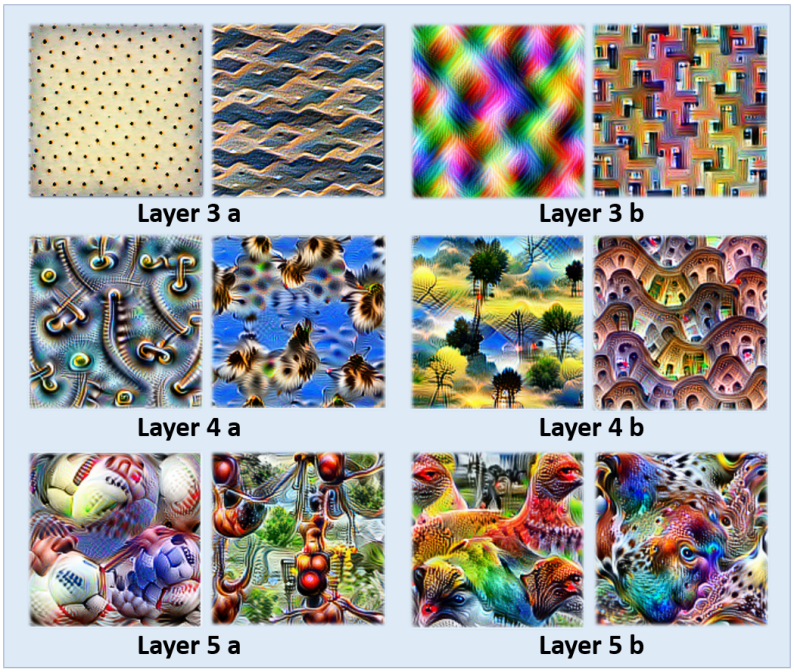

Figure 2.5: Applying the DeepDream algorithm illustrates the progressive complexity of learned features from early layers to intermediate and deeper layers in the GoogLeNet architecture. Adapted from [OMS17].

geous, why not make neural networks arbitrarily deep? The answer lies in the challenges posed by vanishing and exploding gradients destabilizing the training process. To understand this problem, we study the gradients in a simple network.

Its architecture consists of an input layer and T hidden layers (including the output layer) connected to a loss function, denoted $\mathcal{L}$. Each hidden layer consists of a single neuron. The input layer simply transmits the scalar input $h_0 := x$ to the first hidden neuron. Then, given the input h_t the $(t+1)$th neuron computes its output h_{t+1}:

$$\begin{aligned} z_t &= w_{t+1} h_t \\ h_{t+1} &= \Phi(z_t) \ . \end{aligned}$$

We initialize the weights of this network from standard Gaussian distribution, denoted $\mathcal{N}_{\mu=0,\sigma^2=1}$. The local derivative at each layer is

$$\frac{\partial h_{t+1}}{\partial h_t} = \Phi'(z_t) w_{t+1} \ .$$

Setting $\Phi := \mathrm{sigmoid}$ the following relationship holds:

$$\Phi'(z_t) = \Phi(z_t)(1 - \Phi(z_t)) \ .$$

Since sigmoid assums its maximum at 0.5, we have $\Phi'(\cdot) \leqslant 0.25$. Fixing $T = 4$ we can write out the entire derivative of $\mathcal{L}$ with respect to h_1:

$$\frac{\partial \mathcal{L}}{\partial h_1} = \frac{\partial \mathcal{L}}{\partial h_4} \cdot \Phi'(z_3) \cdot w_4 \cdot \Phi'(z_2) \cdot w_3 \cdot \Phi'(z_1) \cdot w_2 \ .$$

In order to fully understand the behavior of the term $\Phi'(z_t)w_{t+1}$ we are interested in the magnitude of the weights w_t. Aggarwal [Agg23] claims that the expected average magnitude of $|w_t|$ is 1 without proof. In my view, this claim is not entirely correct. Since the weights follow the Gaussian standard distribution, we need to compute $Y = |X|$ with $X \sim \mathcal{N}_{0,1}$. In fact, we are interested in the expected value of the *folded normal distribution*. Note that we have $Y = X$ if $X \geqslant 0$ and $Y = -X$ if $X < 0$. This observation allows a straightforward computation:

$$\mathbb{E}[Y] = \int_{-\infty}^{0} \frac{1}{\sqrt{2\pi}} y \exp\left(\frac{-(-y)^2}{2}\right) dy + \int_{0}^{\infty} \frac{1}{\sqrt{2\pi}} y \exp\left(\frac{-(y)^2}{2}\right) dy \tag{2.8}$$

$$= \sqrt{\frac{2}{\pi}} \int_{0}^{\infty} y \exp\left(\frac{-y^2}{2}\right) dy \ . \tag{2.9}$$

Using the method *integration by substitution*, we change variables $z = y^2/2$. Since $z' = y$, we have:

$$\sqrt{\frac{2}{\pi}} \int_{0}^{\infty} \exp(-z) dz = \sqrt{\frac{2}{\pi}} \Big[-\exp(-z) \Big|_{0}^{\infty} \Big] \tag{2.10}$$

$$= \sqrt{\frac{2}{\pi}} \approx 0.79788 \ . \tag{2.11}$$

In conclusion, every hidden layer shrinks the global gradient that flows backward by $\approx 0.25 \cdot \sqrt{\frac{2}{\pi}}$. The drop after ten layers would be in the range of 10^{-7} whereas Aggarwal [Agg23] calculated a drop in the range of 10^{-6}. However, in both cases, we note that earlier layers will receive very small updates compared to later layers. This has severe implications during backpropagation because the parameter updates are proportional to the magnitude of the global gradient. In our network the global gradient decreases exponentially fast. Hence, the updates will also be exponentially small for early layers inhibiting effective training.

One might be tempted to simply replace sigmoid by a function with larger gradients. However, if the gradients are larger than 1 an analogous derivation will lead to the problem of *exploding gradients*. In conclusion, the fundamental problem is that the gradient in early layers is the product of terms from all the later layers. The only way all layers can learn at close to the same speed is if all those products of terms come close to balancing out. This intuition motivates numerous techniques to stabilize gradients as discussed in Sections 3.3 and 3.4.

In this Subsection we studied a simplified example with a single neuron per layer. For the general case, we recall from Subsection 1.3.4 that layer-

to-layer backpropagation updates involve matrix multiplication. Just as in the case of repeated scalar multiplication, it is possible to show that repeated matrix multiplication is inherently unstable. Particularly, eigenvalues of backpropagated gradient matrices greater or smaller than 1 are indicative of unstable gradients. Details can be found in Aggarwal [Agg23].

In summary, vanishing and exploding gradients constitute mathematical challenges that hinder the effective optimization of deep neural networks. For specific neural network architectures (namely recurrent neural networks), formal mathematical proofs confirm that gradients must invariably become unstable during optimization via standard stochastic gradient descent [Hoc91] [BSF94]. The forthcoming Chapter 3 delves into methods and theoretical frameworks designed to alleviate these gradient-related challenges.

3

PRACTICAL CONSIDERATIONS IN NEURAL NETWORK TRAINING

Even though the foundational neural network architecture was established as early as the 1940s [Fit44] and the invention of the backpropagation algorithm in the 1970s sparked brief periods of interest, it wasn't until the algorithmic advancements of the 2010s that the full practical utility of neural networks became evident. This chapter outlines these key developments. Section 3.1 revisits the hyperparameter optimization problem, offering techniques for finding an approximate solution. In Section 3.2, we address the impact of different weight initialization schemes on training dynamics. Afterward, Section 3.3 introduces normalization techniques for layer activations. The last Section 3.4 discusses information highways enabling extremely deep architectures. Even though the techniques presented in this chapter became standard in neural network training, they often lack extensive theoretical analysis. We address this problem by evaluating current research and conducting small experiments to empirically verify the effectiveness of the studied method.

3.1 HYPERPARAMETER OPTIMIZATION

As stated in Subsection 1.2.5 we need to fix an optimal set of hyperparameters $\eta^* \in \hat{\eta}$, here $\hat{\eta}$ denotes the space of all possible hyperparameter configurations, *before the training procedure* starts. Due to the high-dimensional and non-convex nature of this optimization problem, an analytical solution is intractable. Therefore, Subsection 3.1.1 presents several strategies to approximate a reasonable solution. Due to their importance, Subsections 3.1.2 and 3.1.3 separately deal with the learning rate and batch-size hyperparameters. Experiment 3 features a full hyperparameter search by implementing the techniques discussed in the present section.

3.1.1 *Hyperparameter Selection Strategies*

Hyperparameter selection is challenging because different data sets, classification tasks, and neural network architectures produce unique hyperparameter spaces. Furthermore, these hyperparameter spaces exhibit various structures: they can be discrete, as in the case of the number of layers; functional, as in the selection of activation functions; or continuous, as exemplified by the learning rate. Moreover, practitioners often lack a priori knowledge even about the appropriate magnitude for these hyperparameters. We introduce three prevalent methodologies for hyperparameter selection: *grid search*, *multi-resolution sampling* and *random search*.

3.1.1.1 *Grid Search*

The predominant method for hyperparameter optimization is grid search, often employed in conjunction with human intuition. An initial (intuitive) manual search is typically conducted on the high dimensional hyperparameter space to identify promising regions within each single hyperparameter space. Subsequently, the identified regions are partitioned into intervals. A representative value, such as the minimum, maximum, or mean, is selected from each interval, resulting in sets of possible values for each parameter. Grid search necessitates the evaluation of all possible combinations of these hyperparameters. Assuming K hyperparameters, and I_k intervals per hyperparameter the total number of trials, is given by $\prod_{k=1}^{K} |I_k|$. It's worth noting that this approach incurs an exponential computational cost as the number of hyperparameters increases [Bel15]. For instance, having 5 hyperparameters each with 10 possible values results in 10^5 evaluations, requiring the model to be trained $100,000$ times to convergence. While grid search is fully parallelizable, the computational overhead makes it impractical for large-scale tasks, necessitating alternative strategies.

3.1.1.2 *Multi-Resolution Sampling*

One strategy for mitigating the computational expense of grid search is known as multi-resolution sampling [Agg23]. The core idea is to employ a multi-step process. Initially, a coarse grid spanning the entire hyperparameter space is used for sampling. The best hyperparameters identified in this coarse grid serves as the basis for creating a more refined grid. In this second stage, the search is narrowed to a local region centered around these optimal values. This refined search employs a grid that is more granular but smaller in scope than the original grid. The process can be iteratively repeated to focus the search increasingly around the most promising hyperparameters. However, there are drawbacks to this approach. First, it risks prematurely locking the search into a specific region of the hyperparameter space, potentially missing out on other optimal configurations. Second, despite the narrowed focus, the computational cost remains substantial.

3.1.1.3 *Random Search*

In *Random search* one stochastically samples the hyperparameter configurations. In this vein, Bergstra and Bengio [BB12] observe that in practical applications many hyperparameters have little influence on the performance of the model whereas some other hyperparameters are very important. Bergstra and Bengio name this property of the hyperparameter space $\hat{\eta}$ *low effective dimensionality*. The main insight of their research is that random search samples more efficiently from a search space with low effective dimensionality than grid search. We illustrate this intuition with an example adapted from [BB12]. We start by assuming that the function $f(x_1, x_2)$ has a low effective dimensionality and thus can be approximated by another function g, that is $g(x_1) \approx f(x_1, x_2)$.

Because f is mostly sensitive to x_1, the most efficient way to find the minima would be to perform gradient descent on the x_1 axis alone. However, we have to assume that this property of f is initially unknown because in hyperparameter searches the researcher doesn't know in advance which subspace is relevant to the problem at hand. As illustrated on the left side of Figure 3.1 performing grid search for minima of f on the 2-dimensional space covers the whole plane evenly. Moreover, Figure 3.1 demonstrates that the projections of f on the lower dimensional subspaces, that is the x_1-axis and the x_2 axis, are inefficiently covered. In the worst case, the same value of a hyperparameter is repeated in exponentially many configurations.

Compare this to the right side of Figure 3.1 where we sample randomly. We notice that the random points are less evenly distributed in the high-dimensional space. In contrast, more diverse regions in the low-dimensional effective subspace are investigated. By having insight into more diverse regions of the effective low-dimensional subspace we can find a better approximation solution to our minimization problem than with grid search. To confirm their claim, Bergstra and Bengio performed multiple experiments on real-world data sets, revealing that of seven to thirteen only between one to four hyperparameters had a substantial impact on the model's accuracy. Random search often located an effective hyperparameter set within a range of eight to sixteen trials. In contrast, grid search necessitated 256 trials to reach comparable outcomes.

Another advantage of random search is that the computational budget can be chosen independently of the number of parameters and possible values. In consequence, adding hyperparameters does not influence the performance or decrease the computational efficiency of random search. Moreover, random search remains the advantage of being parallelizable like grid search while still allowing for the inclusion of prior knowledge by manually specifying the distribution from which to sample.

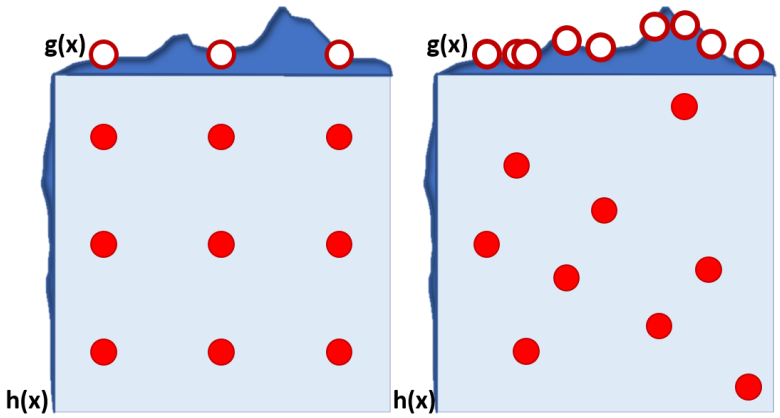

Figure 3.1: The Figure compares the effectiveness of grid (left) and random search (right) when trying to approximate a function $f(x_1, x_2) = g(x_1) + h(x_2)$ with the property $f(x_1, x_2) \approx g(x_1)$. With grid search nine trials test the important function $g(x_1)$ in only three points whereas random search explores g in more diverse regions [BB12].

3.1.1.4 *Logarithmic Sampling*

When we perform random search we need to take the order(s) of magnitude of the search space into account. To elucidate this point, I present we assume that the appropriate number of neurons for a hidden layer falls within the interval $[100, 500]$. In this case, uniform random sampling is a reasonable strategy. Contrast this with the optimization of the learning rate α, where the range of effective values spans several orders of magnitude, for instance $[0.0001, 1]$. If we sampled uniformly from this interval there is a probability $p \approx 0.9$ of sampling within the range $[0.1, 1]$ and only $p \approx 0.1$ for $[0.0001, 0.1]$. This results in an inefficient coverage of the search space, skewed towards higher magnitudes.

An alternative approach is to sample α in the *logarithmic space*, that is in our example in the range $[\log(0.0001), \log(1)] = [-4, 0]$. In consequence, the values $0.0001, 0.001, 0.01, 0.1, 1$ representing the orders of magnitued are evenly spaced. Uniformly sampling in the logarithmic space yields a balanced exploration across magnitudes, with each subinterval having an equal probability $p = 0.25$ of being sampled.

3.1.2 *Learning Rate*

Bergstra and Bengio [BB12] consistently observed that the learning rate was always a critical hyperparameter. This subsection explores advanced strategies like learning rate decay, and visualizes the learning rate's effect on the internal dynamics of early neural network training.

3.1.2.1 *Learning Rate Decay*

At the beginning of training a high learning rate is desirable since the randomly initialized weights of the neural network are unlikely to make good predictions. Additionally, a small learning rate slows down stochastic gradient descent. As the weights are tuned over the course of the training practitioners decay the learning rate because the gradient descent updates typically start to oscillate around a local minimum or even diverge [LKS90]. Recently, researchers suggest an alternative explanation for the effectiveness of learning rate decay: An initially large learning rate suppresses the network from memorizing noisy data whereas a decayed learning rate improves the learning of complex patterns [You+19]. A practice to smoothly anneal the learning rate is *exponential decay*:

$$\alpha_t = \alpha_{start} \exp(-k \cdot t) \, ,$$

where α_{start} denotes the start learning rate, t the training step or epoch and $k \in \mathbb{R}$ is a hyperparameter fixed before training. Experiment 3 applies this technique with a self-derived modification that avoids the introduction of the hyperparameter k. In my implementation, I randomly sample the decay factor $m << 0.1$. Then I decrease the learning rate from α_{start} to

$m \cdot \alpha_{start}$. In my view, this is more consistent with the approach of random hyperparameter sampling [BB12]. Another possible approach for learning rate decay was used by Bengio [Ben+03] in their language model is referred to as *inverse decay* defined by

$$\alpha_t = \frac{\alpha_{start}}{1 + k \cdot t} .$$

All methodologies have in common, that an initially high learning rate quickly decreases. In consequence, a majority of the learning steps have a very low size compared to the starting learning rate. The optimal learning rate is usually close to the largest learning rate that does not cause divergence of the training loss [Ben12].

3.1.2.2 *Visualizing Learning Rate Dynamics*

Crucially, even a slightly inappropriate value for α at the onset of training prevents the model from learning: Too high a learning rate causes the model parameters to oscillate or diverge from the start, while too low a learning rate prevents the model from ever converging. This subsection offers a technique to visually comprehend the effect of the learning rate on the parameter updates and thus pick a reasonable value for α. For this purpose, we introduce a metric known as UD_t, adapted from Glorot and Bengio [GB10]. This metric captures the relative amount the learning parameters are changed after t updates via backpropagation. Given a connection matrix W corresponding to a specific layer in the neural network its UD_t metric is mathematically expressed as:

$$UD_t := \log\left(\frac{\alpha_t \cdot \sigma(\partial\theta_t)}{\sigma(\theta_t)}\right) .$$

Here, θ_t denotes the individual learning parameter values in W at time step t and $\partial\theta_t$ their derivatives. Furthermore, we compute the standard deviation across all individual parameter values in W, denoted $\sigma(\theta_t)$, and their respective gradients, denoted $\sigma(\partial\theta_t)$.

To illustrate how the UD_t metric can serve as a tool for finding an appropriate learning rate in a principled manner, I have conducted Experiment 4. In this experiment we capture the UD_t metric for the first $1,000$ learning steps for all linear layers of three models. These three models each consist of the exact same architecture however they are initialized with a high, medium and low learning rate. Figure 3.2 reveals that the *high learning rate* model exhibits pronounced oscillations impeding convergence. Moreover, we observe a hierarchical learning pattern, where later layers learn faster than the early ones. Such behavior indicates that the full capacity of the model is not used. For the *low learning rate* scenario the UD_t value of around -12 indicates that the learning speed is *several magnitudes lower* than in the other scenarios suggesting that the learning rate is too conservative. In contrast, the *medium learning rate* model achieves a reasonable learning pace, as reflected by the UD_t values approximating -6 for most layers, devoid of pronounced oscillatory behaviors. This balanced learning is advantageous, ensuring that both,

early and late layers, play a meaningful role in feature extraction and representation. By monitoring the UD_t values for every layer during the initial stages of training, we conclude that the medium learning rate strikes the best balance between stability and convergence speed.

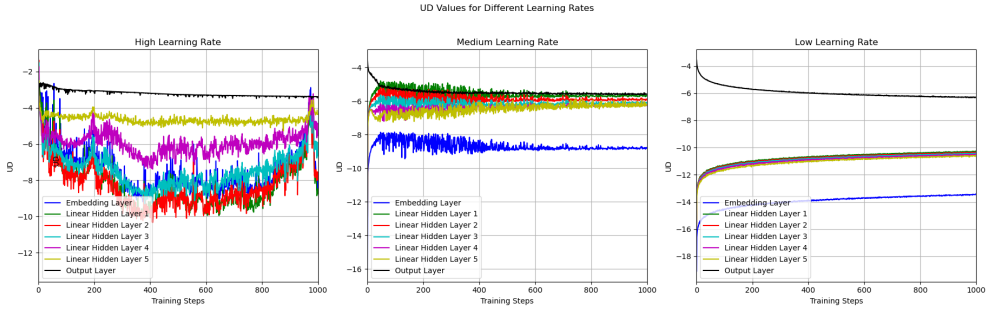

Figure 3.2: The figure features the learning dynamics of three neural networks differing only in their learning rate. The high learning rate leads to oscillation in learning parameter updates, a low learning rate to slow convergence whereas the medium learning rates strikes a balance between learning speed and stability.

3.1.3 *Mini-Batch Size*

This subsection introduces another hyperparameter called *mini-batch size* essential to accelerating the gradient descent during backpropagation. Setting the mini-batch size to 1 leads to online learning already examined in Example 1.

3.1.3.1 *Point-Wise Stochastic Gradient Descent*

In online learning, we update the weights, denoted θ, from the set of all learnable parameters in a neural network, denoted Θ iteratively based on individual training instances $(\vec{x}, \vec{y})$ randomly sampled from an assumed data generation source q:

$$\theta \leftarrow \theta - \alpha \frac{\partial \mathcal{L}(f(\vec{x}; \Theta), \vec{y})}{\partial \theta} .$$

This updating mechanism, termed as *point-wise stochastic gradient descent* often leads to volatility in the training process and erratic updates on the loss landscape. The suggested reason for this behavior is the noise inherent in individual data points due to redundancies in the data set, mislabeling, etc. [Agg23].

3.1.3.2 *Gradient Descent*

The opposite extreme is to base updates on the entire training set, called gradient descent: Here, the objective of a single backpropagation update is to minimize the loss $\hat{\mathcal{L}}$ on the entire training set $\mathbb{X}^{train}$ defined by:

$$\hat{\mathcal{L}} := \sum_{(\vec{x},\vec{y})\in\mathbb{X}^{train}} \mathcal{L}(f(\vec{x};\Theta),\vec{y}) \,. \tag{3.1}$$

To apply gradient descent, one computes the gradient of the loss of the entire data set with respect to individual weights $\frac{\partial\hat{\mathcal{L}}}{\partial\theta}$. However, in order to evaluate this derivative we need to forward and then backpropagate the entire data set through the network. Since one would have to store all intermediate activations and derivatives for each training instance for all neural network nodes for a single update step this method is intractable for deep neural networks. Moreover, this technique is not only impractical but also mathematically unjustified: In fact, the gradient only locally indicates the steepest descent direction. If we wanted to perform larger update steps we should incorporate information from the second derivative. Additionally, the gradient for the entire training set only *estimates* the true gradient. As examined in Subsection 2.2.1.1, the true gradient direction corresponds to the *expected gradient* over all possible training instances, weighted by the ground truth distribution q. The intuition that it is more advantageous to consider the computational budget than to determine the mini-batch size based on the time spent collecting the data set leads to mini-batch stochastic gradient descent.

3.1.3.3 *Mini-Batch Stochastic Gradient Descent*

Mini-batch stochastic gradient descent serves as a compromise. This approach performs weight updates based on an average gradient from a batch $B = \{(\vec{x}_{j_1},\vec{y}_{j_1}),\ldots,(\vec{x}_{j_m},\vec{y}_{j_m})\}$, consisting of m randomly selected training examples:

$$\theta \leftarrow \theta - \frac{\alpha}{m}\cdot\sum_{(\vec{x},\vec{y})\in B}\frac{\partial\mathcal{L}(f(\vec{x};\Theta),\vec{y})}{\partial\theta} \,. \tag{3.2}$$

The rationale is that training data often contains a high degree of redundancy and thus mini-batch stochastic gradient descent acts as a form of implicit regularization by averaging out individual data point noise [Agg23]. Moreover, it is more advantageous to take more frequent but less accurate steps using mini-batches enabling a computational more efficient and broader exploration of the loss landscape compared to fewer, presumably more accurate, updates based on the entire dataset. A recent study confirms that in practice mini-batch stochastic gradient descent outperforms the previous techniques and leads to good generalization if the mini-batch size is not overly large [Kes+17].

As the previous study [Kes+17] examines in more detail, the choice of mini-batch size is subject to a complex interplay of factors such as hardware

limitations, dataset size, and the search for optimal model generalization. Specifically, memory constraints imposed by hardware impose a natural upper limit on feasible mini-batch sizes whereas computational efficiency advocates for larger mini-batch sizes, which are more amenable to parallel hardware-optimized matrix operations. Balancing these competing factors, a mini-batch size that prioritizes computational efficiency within the bounds of hardware limitations often emerges as a pragmatic choice. Commonly used values are powers of 2 (such as $32, 64, 128, 256$) as they can be efficiently implemented on the hardware level [Agg23].

3.2 INITIALIZATION

This section offers a theoretical treatment and also a practical guideline for weight initialization: Subsection 3.2.1 discusses *heuristic initialization* and how this scheme causes *variance instability* across layers. In Subsection 3.2.2, we formally derive *Xavier initialization*, designed to stabilize variance. Subsection 3.2.3 studies the contraction effects of certain activation functions and formally derives a technique to counteract this problem at initialization time.

3.2.1 *Heuristic Initialization*

Before 2010 neural networks were initialized heuristically with weights generated from a Gaussian distribution $\mathcal{N}(0, \sigma^2)$ where $\sigma^2 \approx 10^{-2}$ [Ben+03] [GB10]. However, this naive approach leads to several problems. First, heuristic initialization does not take the number of inputs each neuron receives into account. As a result, neurons with fewer inputs tend to be more sensitive to changes in their individual inputs compared to neurons with a large number of inputs. This sensitivity disparity creates imbalances during training, as neurons with fewer inputs dominate the learning process [Agg23]. The second problem of heuristic initialization is related to the variance of the output activations across layers. In this vein, Aggarwal [Agg23] argues that the output variance for a neuron scales linearly with the number of its inputs leading to vanishing or exploding gradients. In Lemma 2, I prove his claim through mathematical analysis.

Lemma 2 (Linear Scaling of Output Variance) *Let $x_1, \cdots, x_n$ and $w_1, \cdots, w_n$ be i.i.d. random variables sampled from a zero-centered Gaussian with unit variance representing the inputs and weights of a single neuron, respectively. Then, the variance of the pre-activation output z of the neuron scales linearly with the number of inputs.*

Proof We can express the pre-activation output z as a sum of the weighted inputs:

$$z = \sum_{i=1}^{n} w_i x_i$$

Next, we compute the variance of this pre-activation and apply the properties of the random variables:

$$\begin{aligned}
\mathrm{Var}(z) &= \mathrm{Var}\left(\sum_{i=1}^{n} w_i x_i\right) \\
&= \sum_{i=1}^{n} \mathrm{Var}(w_i x_i) \\
&= \sum_{i=1}^{n} \left(\mathbb{E}[w_i]^2 \mathrm{Var}(x_i) + \mathbb{E}[x_i]^2 \mathrm{Var}(w_i) + \mathrm{Var}(w_i)\mathrm{Var}(x_i)\right) \\
&= n
\end{aligned}$$

∎

When the preactivation value of a neuron has a linearly scaling variance, it implies that the inputs across layers grow large in absolute terms in an uncontrollable manner. Since many activation functions such as the tanh saturate for extremely negative or positive inputs, high variance in the preactivation can lead to saturation of the activation function. As these functions saturate the gradient vanishes during backpropagation.

In direct relationship to the notion of linearly scaling variances of activations, Glorot and Bengio [GB10] verified through investigative experiments that the heuristic initialization scheme consistently leads to excessive saturation of activation functions. Experiment 5 validates this analysis: In this experiment the model comprises four fully connected layers, each followed by a tanh activation. The weights were heuristically initialized from a Gaussian with zero mean and 0.1 standard deviation. A batch of 500 random samples serves as input to the network, and activations from each layer are recorded and analyzed. The results shown in Figure 3.3 indicate that the activations in the first layer are highly saturated, gravitating towards -1 and 1. This effect quickly accumulates across the layers leading to an extremely saturated regime in the last layer. This situation directly after initialization effectively nullifies gradients during backpropagation and agrees with the results found by Glorot and Bengio [GB10].

3.2.2 *Xavier Initialization*

To maintain a consistent variance for activations across layers, Xavier Glorot and Yoshua Bengio proposed a novel weight initialization scheme, commonly known as Xavier Initialization [GB10]. This widely used method rests on several simplifying assumptions [GB10]:

- Weights and inputs are i.i.d. distributed from a zero-centered Gaussian with unit variance.
- Biases are initialized as zeros.

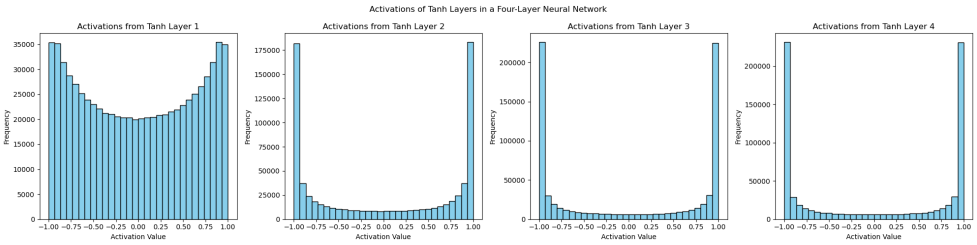

Figure 3.3: Distributions of tanh activation values across four layers of a neural network initialized with heuristic weights from $\mathcal{N}(0, 0.1)$. The x-axis represents the activation value, ranging from -1 to 1, and the y-axis indicates the frequency of neurons producing a given activation. The activations progressively saturate towards -1 and 1 as layers increase, exemplifying the vanishing gradient problem.

- The tanh activation function is used, which is approximately linear for small input values.

To maintain a constant variance of activations across layers, we examine how to initialize the weights, specifically setting $\text{Var}(W^l[i,j])$ where W^l is the $p^l \times p^{l-1}$ connection matrix between the $(l-1)$th and l-th layer. We denote the post-activation vector of the l-th layer as $\vec{h}^l$ and its preactivation vector as $\vec{z}$. We begin by equating the variance of the activation for a neuron i in layer l, denoted $\text{Var}(\vec{h}^l[i])$, to the variance of the activation for a neuron i in the preceding layer $l-1$, denoted $\text{Var}(\vec{h}^{l-1}[i])$:

$$\begin{aligned}
\text{Var}(\vec{h}^{l-1}[i]) &= \text{Var}(\vec{h}^l[i]) && (3.3)\\
&= \text{Var}(\vec{z}^l[i]) && (3.4)\\
&= \text{Var}\left(\sum_{j=1}^{p^{l-1}} W^l[i,j]\vec{h}^{l-1}[j]\right) && (3.5)\\
&= \sum_{j=1}^{p^{l-1}} \text{Var}\left(W^l[i,j]\vec{h}^{l-1}[j]\right) && (3.6)\\
&= \sum_{j=1}^{p^{l-1}} \Big(\mathbb{E}[W^l[i,j]]^2\text{Var}(\vec{h}^{l-1}[j]) \\
&\qquad + \mathbb{E}[\vec{h}^{l-1}[j]]^2\text{Var}(W^l[i,j]) + \text{Var}(W^l[i,j])\text{Var}(\vec{h}^{l-1}[j])\Big) && (3.7)\\
&= p^{l-1}\text{Var}(W^l[i,j])\text{Var}(\vec{h}^{l-1}[i]) && (3.8)\\
\implies \text{Var}(W^l[i,j]) &= \frac{1}{p^{l-1}} && (3.9)
\end{aligned}$$

- Equation (3.4) uses the linearity of the tanh function around zero.
- Equation (3.5) applies the variance of an independent sum.

- Equation (3.6) uses the definition of a preactivation.
- Equation (3.7) employs the formula for the variance of independent products.
- Equation (3.8) simplifies the equation under the assumption of identical distributions.

In conclusion, Xavier initialization offers a methodical way to initialize the weights of neural networks such that the variance of activations remains consistent across layers:

$$W^l[i,j] = \mathcal{N}\left(0, \frac{1}{p^{l-1}}\right) .$$

Remark 6 (Zero Weight Initialization) One might be tempted to sidestep the complexities of weight initialization by simply zeroing out all weights. However, the inherent symmetry in most neural architectures causes gradient descent updates to propagate uniformly across the network. As a result, neurons within the same layer become almost indistinguishable in behavior, effectively throttling the network's expressivity. In contrast, Bengio [Ben12] recommends initializing the biases to zero.

3.2.3 *Mitigating Vanishing Activations*

Independently from mitigating the effect of the different sizes of layers, we must also deal with the influence of the specific activation function on the variance of the activations. This section focuses on the tanh function's contraction effect. In general, the mathematical analysis of this section can be adapted to control analogous effects of other activation functions.

3.2.3.1 *Understanding Vanishing Activations*

Like most activation functions, the tanh function contracts values greater than 1 in absolute terms to the open interval $(-1, 1)$. Moreover, the function contracts also values in $(-1, 1)$ to even smaller values. In a multi-layer neural network architecture where tanh is employed as the activation function at each layer, the composition $\tanh \circ \tanh \circ \cdots \circ \tanh(\cdot)$ gives rise to a compounded contractive effect. This forces the activation values to converge towards zero as one advances through the network layers making the forward propagation numerically unstable.

3.2.3.2 *Deriving the Gain*

To address the challenge of vanishing activations, the *gain* — a scaling factor applied to the weights during initialization — serves as a countermeasure. To quantify the relative impact of tanh activations, I calculate the variance of the function's output given an input sampled from $x \sim \mathcal{N}(0, 1)$:

$$\mathrm{Var}(\tanh(x)) = \mathbb{E}[\tanh(x)^2] - \mathbb{E}[\tanh(x)]^2$$

Because $\tanh(x)$ satisfies $\tanh(-x) = -\tanh(x)$, and x is symmetrically distributed around zero, it follows $\mathbb{E}[\tanh(x)] = 0$. This simplifies the variance expression to $\mathrm{Var}(\tanh(x)) = \mathbb{E}[(\tanh(x))^2]$. The expected value $\mathbb{E}[(\tanh(x))^2]$ can then be computed using the integral:

$$\mathbb{E}[(\tanh(x))^2] = \int_{-\infty}^{\infty} (\tanh(x))^2 \frac{e^{-\frac{x^2}{2}}}{\sqrt{2\pi}} \, dx$$

Utilizing numerical computation via the scipy library [Vir+20], the integral for this expected value approximates to 0.39. Taking the square root yields $\sqrt{0.39} \approx \frac{3}{5}$, indicating the factor by which the variance of the incoming variable is reduced through the tanh function. Thus, this effect can be counterbalanced by multiplying the weights with its reciprocal, $\frac{5}{3}$, to preserve unit variance in the output activations. In sum, I justified the gain factor given in the PyTorch library without mathematical analysis [Pas+17].

In summary, the application of Xavier initialization fine-tuned with a gain factor serves as a countermeasure to the issues arising from instable activations. Given the results of this subsection, we initialize the weights of linear layers that are followed by a tanh acvativation according to a Gaussian distribution $\mathcal{N}(0, \frac{5}{3} \cdot \frac{1}{p^{l-1}})$ where p^{l-1} denotes the number of neurons of the previous layer.

3.2.3.3 *Visualizing the Benefit of Fine-Tuned Xavier Initialization*

To visualize the benefit of the Xavier initialization fine-tuned with a gain we rerun experiment 5 from Subsection 3.2.1. However, this time we initialize the network according to our mathematical derivations. Figure 3.4 obtained from the second part of experiment 5 depicts the activations which appear Gaussian and centered around zero. This optimizes tanh function's gradient close to 1, thus ensuring effective error backpropagation from the start of training. In conclusion, the choice of initialization has substantial implications for the neural network's training efficiency. While heuristic initialization leads to instable gradients and activations, the fine-tuned Xavier initialization provides a conducive environment for gradient flow, enhancing training efficiency.

3.2.4 *Setting up the Softmax Layer*

In this subsection, we delve into three aspects of the softmax output layer essential for efficient training: *lateral inhibition*, *numerical stability*, and *confidence levels*.

3.2.4.1 *Lateral Inhibition*

In neurobiology, *lateral inhibition* is the capacity of an excited neuron to reduce the activity of its neighbors. In the deep learning literature it is often argued that the softmax output layer encourages a form of approximate acti-

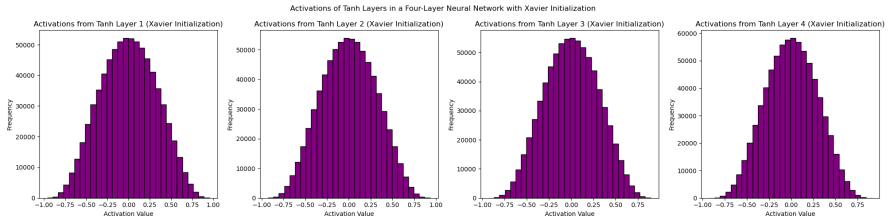

Figure 3.4: Distribution of tanh activation values across four consecutive layers in a neural network, initialized using the Xavier method. The x-axis represents the activation value, ranging from -1 to 1, while the y-axis indicates the frequency of each activation value. The Gaussian-like distributions centered around zero demonstrate the effectiveness of the Xavier initialization in maintaining optimal activations and preventing vanishing gradients across layers.

vation sparsity, also termed *co-occurrence sparsity*. For the softmax this means that in any particular instance relatively few neurons are highly activated [Elh+22]. In this section, we mathematically justify that the softmax layer in neural networks mimics this lateral inhibition. To mathematically demonstrate this intuitive view on the softmax layer I introduce the following ratio: Given a vector of inputs $\vec{x} = (x_1, \cdots, x_n)$, setting $x_{max} := \max(x_1, \cdots x_n)$ and applying the softmax notation convention (Equation 1.5) we obtain:

$$\frac{s_i(\alpha \cdot \vec{x})}{s_{max}(\alpha \cdot \vec{x})} = \exp(\alpha(x_i - x_{max})) \, . \tag{3.10}$$

From this equation, we can make two key observations:

1. When $x_i = x_{max}$, the ratio is 1, signifying that the neuron with the maximum activation retains its high level of activity.

2. For $x_i < x_{max}$, the ratio exponentially converges to zero as the scaling factor α increases, thereby suppressing the activations of neurons with lower initial activations.

In conclusion, scaling the inputs to a softmax function in a linear fashion suppresses all output neurons except of those that correspond to the maximum input *in an exponential manner*. This leads to a sparse output vector, concurring with the view that the softmax layer encourages lateral inhibition. Understanding the softmax function's sensitivity to scaling is a prerequisite for examining confidence levels (Subsection 3.2.4.2).

3.2.4.2 *Confidence Levels*

At the beginning of neural network training practitioners often observe a huge loss which dramatically decreases during the first few training steps. This is an artifact of initializing the weights before the softmax heuristically: To illustrate this, suppose we have four classes, and the softmax function is plugged into the cross-entropy loss. Moreover, let the input of the softmax be $(2, 5, 4, -5)$. In consequence the softmax outputs probabilities

$\approx (0.04, 0.70, 0.26, 0)$. In fact, lateral inhibition pushes the softmax from the start to be overly confident in assigning an unjustified high probability to one - here the second - of the classes. In settings with many classes, the prediction after random initialization is usually wrong. Assuming that the ground truth class is the first we compute a loss of $-\log_2(0.04) \approx 4.64$. Compare this to the uniform distribution over the four classes where the negative log-likelihood would be $\log_2(0.25) = 2$. In conclusion, unscaled inputs lead to a skewed probability distribution with high confidence in random classes which is most of the times *worse than a random guess*.

To ensure that from the start of training, our model is at least as good as a random guess we push the softmax to be less confident. We achieve this by scaling the inputs with a small factor (for instance 0.1). In our example, the inputs are scaled down to $(0.2, 0.5, 0.4, -0.5)$ so that the softmax initially computes probabilities much closer to the uniform distribution: $\approx (0.25, 0.33, 0.30, 0.12)$.

3.2.4.3 *Visualizing Confidence Levels*

Figure 3.5 visualizes various confidence levels resulting from Experiment 6. In this experiment, we forward propagate randomly sampled training instances through a simple feed-forward model and collect their post-softmax activations. Figure 3.5 displays three histograms: one for unscaled weights before the softmax and two others for scaling factors of 0.1 and 0.05, respectively. The unscaled softmax activations depict a pronounced peak and a wider distribution, revealing the model's tendency to confidently assign predictions to specific classes upon random initialization. This overconfidence often misaligns with the true class. The scaled input histograms, however, demonstrate more evenly spread and much narrower distributed activations. In fact, the histograms resemble a uniform distribution in class assignment. In conclusion, input scaling is an important strategy to temper initial softmax outputs leading to a reduced initial loss and more efficient training

3.2.4.4 *Numerical Stability*

In contrast to the sensitivity to scaling, Lemma 3 reveals the softmax layer's resilience to translation:

Lemma 3 *The softmax layer is indifferent to translation with a factor* $m \in \mathbb{R}$ *of its inputs variables* $(x_1, \cdots, x_n)$*, or more precisely for every* k *with* $1 \leqslant k \leqslant n$ *we have,*

$$\frac{\exp x_k}{\sum_{i=1}^{n} \exp x_i} = \frac{\exp(x_k + m)}{\sum_{i=1}^{n} \exp(x_i + m)}$$

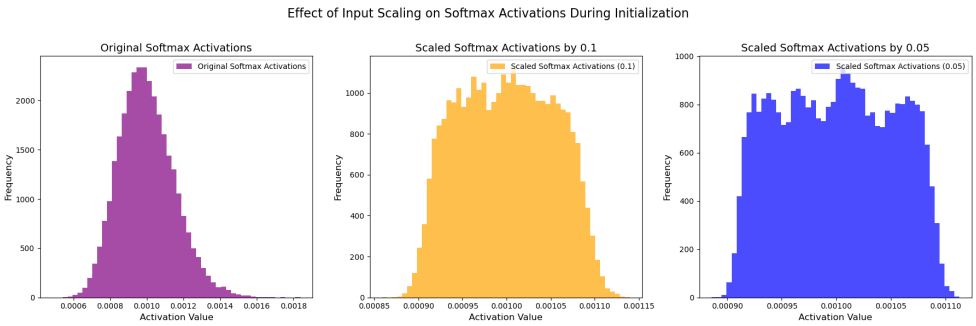

Figure 3.5: Comparison of Softmax Activation Distributions for Different Input Scales. The x-axis represents the predicted probability values of softmax outputs, ranging from approximately 0 to 0.002. The y-axis depicts the frequency with which these values occur. The leftmost histogram showcases the distribution for unscaled inputs, with a prominent peak around 0.0010. The middle and rightmost histograms detail the distributions for inputs scaled by 0.1 and 0.05, respectively. As scaling factors decrease, the distribution broadens, reflecting less confident class assignments and a trend toward uniformity.

Proof We compute in a straightforward way,

$$\frac{\exp(x_k + m)}{\sum_{i=1}^{n} \exp(x_i + m)} = \frac{\exp x_k \exp m}{\sum_{i=1}^{n} \exp(x_i) \exp m}$$
$$= \frac{\exp x_k \exp m}{\exp m \sum_{i=1}^{n} \exp(x_i)} = \frac{\exp x_k}{\sum_{i=1}^{n} \exp x_i}$$

∎

Let's now turn our attention to the numerical stability of the softmax function. To illustrate, consider the logit vector of $(4, 2, 987, 0)^\intercal$ that the softmax has to normalize. Calculating $\exp 987$ results in numerical overflow. To mitigate this, it is standard practice to subtract the maximum value from all the logits [Ben+03]. In our example, the modified input to the softmax would be $(4 - 987, 2 - 987, 987 - 987, 0 - 987)^\intercal$ leading to numerical stability. By Lemma 3, this translation leaves the output probabilities unchanged.

3.3 NORMALIZATION TECHNIQUES

This section introduces two techniques aiming to mitigate *internal covariate shift* discussed in Subsection 3.3.1: *batch normalization* and *layer normalization*. Batch normalization facilitates deeper networks, allows for a higher learning rate and reduces the network's sensitivity to initialization [IS15]. For the first time in human history, batch normalization, presented in Subsection 3.3.2, enabled a neural network to outperform human raters in ImageNet, a famous object recognition challenge [IS15]. However, batch norm introduces batch-related side effects and is difficult to transfer to language processing architectures. Layer normalization, discussed in Subsection 3.3.3, addresses these problems and became the choice of normalization in the ground break-

ing transformer architecture presented in Chapter 4. The last Subsection 3.3.4 is devoted to visualize the stabilizing effect of the normalization techniques on gradients during backpropagation.

3.3.1 *Internal Covariate Shift*

Whereas current deep learning textbooks [Agg20] [Agg23] [GBC18] only briefly touch upon the phenomenon of internal covariate shift, I aim to describe this problem more precisely. We focus on the j-th neuron in layer $l+1$. The pre- and post-activation $\vec{z}^{l+1}[j]$, and $\vec{h}^{l+1}[j]$ for this neuron are calculated as:

$$\vec{z}^{l+1}[j] = W^{l+1}[j,:]\vec{h}^{l} \tag{3.11}$$

$$\vec{h}^{l+1}[j] = \Phi(\vec{z}^{l+1}[j]) \, . \tag{3.12}$$

Here, $W^{l+1}[j,:]$ represents the j-th row of the $p^{l+1} \times p^{l}$ connection matrix, $\vec{h}^{l}$ the post-activation vector of layer l, $\vec{h}^{l}[j]$ its j-th coordinate and Φ the activation function. Equations 3.11 and 3.12 demonstrate that as the distribution of $\vec{h}^{l}$ changes due to training updates in weight parameters $\mathbf{W}^{k}$ for $k < l+1$ the input distribution of both $\vec{z}^{l+1}[j]$ and $\vec{h}^{l+1}[j]$ change. This change is termed internal covariate shift.

Mean shift and *variance shift* are examples of such changes. With regard to a single neuron, *mean shift* refers to how a change in the distribution of $\vec{h}^{l}$ alters the expected value $\mathbb{E}[\vec{z}^{l+1}[j]]$, affecting the point where the activation function Φ is evaluated. *Variance Shift* refers to how the variance $\mathrm{Var}(\vec{z}^{l+1}[j])$ changes, affecting the spread of the activation function [IS15]. This shifting landscape has several implications: The non-stationary nature of the optimization landscape blurs already learned patterns in later layers, and necessitates lower learning rates and more careful initialization. Additionally, rapid changes in the internal activation distributions are observed to lead to numerical instabilities, and vanishing or exploding gradients [IS15].

3.3.2 *Batch Normalization*

Batch normalization mitigates internal covariate shift by normalizing the input of each layer for each batch. This keeps the input distributions consistent during training, making it easier for later layers to learn and represent patterns in the data.

3.3.2.1 *Technical Details*

Suppose we forward propagate in parallel a batch of m training instances through our neural network. Let $\vec{z}^{r}[i]$ be the preactivation value of the i-th

neuron of a given layer for the r-th instance of the given batch. First, we compute *batch- statistics* for every neuron:

$$\mu_i = \frac{\sum_{r=1}^{m} \vec{z}^r[i]}{m}$$
$$\sigma_i^2 = \frac{\sum_{r=1}^{m} (\vec{z}^r[i] - \mu_i)^2}{m} .$$

Subsequently, we normalize all m instances of the batch corresponding to the i-th hidden unit and scale them with the node-specific batch norm parameters $\beta_i, \gamma_i \in \mathbb{R}$:

$$\vec{z}^r[i] \leftarrow \frac{\vec{z}^r[i] - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}} \tag{3.13}$$
$$\vec{z}^r[i] \leftarrow \gamma_i \cdot \vec{z}^r[i] + \beta_i . \tag{3.14}$$

After normalization and scaling, the pre-activation of the i-th neuron assumes a mean of β_i and a standard deviation of γ_i. These parameters offer the model increased adaptability in approximating the data-specific distribution and are trained during backpropagation, similar to the other model parameters. The ϵ term in Equation 3.13 ensures numerical stability by preventing division by zero. Subsequently, the modified pre-activation values serve as inputs to the activation functions. It's worth noting that all computations can be performed in parallel for all neurons in a given layer, thereby improving computational efficiency.

Curiously, the randomly selected instances of a single batch influence each other's normalization, an effect that, while unintended, has been found to act as a form of regularization [Iof17]. However, this regularization effect diminishes when the batch size is small or the samples in the mini-batch are not independent [WH18]. Thus, batch normalization introduces a new trade-off: Whilst a large batch size yields more reliable batch statistics it risks undermining the advantages of mini-batch stochastic gradient descent over traditional gradient descent.

3.3.2.2 *Batch Norm During Inference*

When performing inference or testing, the batch size could be as small as one, making it impossible to compute batch statistics. To address this, we maintain running averages for both the mean, denoted $\hat{\mu}_i$, and the variance, denoted $\hat{\sigma}_i^2$, which are updated during training. The updates are:

$$\hat{\mu}_i = M * \hat{\mu}_i + (1 - M) * \mu_i$$
$$\hat{\sigma}_i^2 = M * \hat{\sigma}_i^2 + (1 - M) * \sigma_i^2$$

Here, $M \in [0, 1]$ represents the *momentum*, a hyperparameter usually set close to 1. The momentum balances the influence of the statistics of the current mini-batch with those of the previous mini-batches. A natural idea

is to use these running averages for normalization during training. However, in a subsequent paper, Ioffe [Iof17] demonstrates that this approach leads to the model's parameters becoming extremely large.

3.3.3 *Layer Normalization*

Batch normalization has limitations, particularly its dependency on a minimum batch size for model accuracy [WH18]. Techniques like *layer normalization* [BKH16], *group normalization* [WH18], and *instance normalization* [UVL17] were proposed as alternatives. We restrict this section to the discussion of layer normalization because it is the normalization used in the transformer architecture (Chapter 4). In contrast to batch normalization, layer normalization does not rely on batch statistics. Instead, it normalizes activations for each individual instance across all neurons. This distinction is illustrated in Figure 3.6.

More precisely, let $\vec{z}^{\,r}[i]$ be the preactivation value of the i-th neuron of the l-th layer containing p^l units for the r-th instance of a batch. For instance we compute the mean μ^r and the standard deviation σ^r over all p^{l+1} units:

$$\mu^r = \frac{\sum_{i=1}^{p^{l+1}} \vec{z}^{\,r}[i]}{p^{l+1}}, \qquad \sigma^r = \sqrt{\frac{\sum_{i=1}^{p^{l+1}} (\vec{z}^{\,r}[i] - \mu^r)^2}{p^{l+1}} + \epsilon}\,. \tag{3.15}$$

Similarly to batch norm, every neuron contains a gain parameter γ^i and a bias parameter β^i which are used to scale the normalized values and add a bias.

$$\vec{z}^{\,r}[i] \leftarrow \frac{\gamma^i}{\sigma^r} \cdot (\vec{z}^{\,r}[i] - \mu^r) + \beta^i\,.$$

Afterward, the activation function Φ is applied to the scaled and normalized values. Analagous to batch normalization, the gain and bias parameters give the model more flexibility and are updated during backpropagation. Interestingly, recent research indicates that these parameters can lead to overfitting and may not be universally effective [Xu+19]. Whereas batch normalization is favored in image recognition, layer normalization has shown greater efficacy in neural language prediction tasks [Xu+19].

3.3.4 *Visualizing the Benefits of Normalization Techniques*

By visualizing gradient distributions in a layer-wise fashion practitioners diagnose learning inefficiencies such as vanishing or exploding gradients [GB10]. In experiment 7, we study a ten-layer feed-forward neural network that utilizes stochastic gradient descent and a cross-entropy loss function for training. To focus on early training dynamics, the training is halted after 1,000 iterations, and the gradients of the layers aare collected. In practical settings, this is a typical procedure to analyze and debug the chosen architecture before committing to extensive training. Three architectures are con-

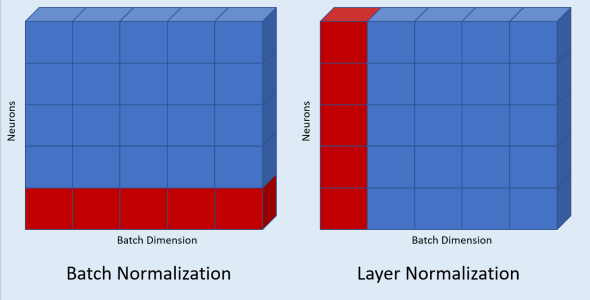

Figure 3.6: In batch norm, the mean and variance used for normalization are calculated across all instances in a batch, for each feature independently. For layer norm, the statistics are calculated across neurons, for each batch instance independently.

sidered: a baseline model, and variants with batch and layer normalization respectively.

Refer to Figure 3.7 for a log-scaled view of these gradient distributions. In the baseline model, gradients in the embedding layer are exceptionally high, indicative of exploding gradients, while subsequent layers show near-zero gradients, pointing to vanishing gradients. Both normalization variants, however, maintain gradients within a narrow range, mitigating these extreme behaviors. These results are in line with the literature that both, layer and natch normalization succeed in regulating the gradients within a reasonable range which facilitates balanced learning throughout the network.

3.4 INFORMATION HIGHWAYS

Feed-forward neural networks, characterized by each layer i directly feeding its output to layer $i+1$, have a common challenge: vanishing and exploding gradients, a problem only partially solved by the activation normalization techniques. To address this problem, *skip connections* that bypass certain layers have been explored early on [Ben+03] [RVL12] [Gra14]. Srivastava et al. conceptualized this architectural tweak and referred to this concept as *information highway* [SGS15]. Subsection 3.4.1 examines the potential of highway networks in stabilizing gradients. Subsection 3.4.2 discusses a related idea, termed *residual connections*. Finally, Subsection 3.4.3 visualizes the benefits of residual connections in an extremely deep network.

3.4.1 *Deep Highway Networks*

As a historical side note, Srivastava discovered highway networks when working at a research lab headed by Schmidhuber at IDSIA. Schmidhuber essentially applied this idea already in the 90's on reccurrent neural networks, thus, inventing the *long short term memory networks* (LSTM) which dominated neural language processing before the invention of the transformer architecture. In fact, LSTMs could be viewed as a special instance of highway

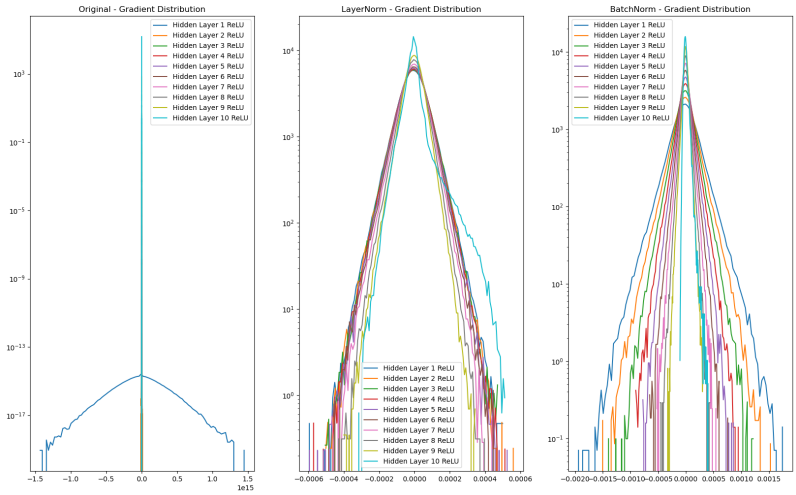

Figure 3.7: Log-scaled gradient distributions of ten hidden layers with ReLU activations in three neural network architectures: the baseline, layer normalization, and batch normalization. The x-axis represents the gradient values, while the y-axis indicates the frequency of these values. In the baseline, the first layer manifests exploding gradients, while the subsequent layers display vanishing gradient behavior. Contrarily, with layer and batch normalization, the gradient values are consistently maintained within a bounded range across all layers.

networks. Importantly, highway networks allow for effectively training neural network architectures of unprecedented depths, ranging from hundreds to even a thousand layers [SGS15] [He+15]. Intuitively, highway networks implement switchable skip connections through gate units. These trainable gate units enable and regulate selective information flow from the previous to the subsequent layer.

3.4.1.1 *Technical Details*

While Srivastava et al. [SGS15] describes highway networks in a layer-wise fashion, I focus on the computational dynamics of a single neuron since this perspective allows for a more rigorous and straightforward analysis: For the i-th neuron in the l-th layer, we define the non-linear transformation g_i. In a basic neural network, given the input $\vec{h}^{l-1}$, the neuron's function g_i applies a linear transformation parameterized by a weight matrix W_g^l followed by a nonlinear activation function Φ:

$$\vec{h}^{l}[i] = g_i(\vec{h}^{l-1}) := \Phi(W_g^l[i,:]\vec{h}^{l-1}) \, .$$

In highway networks, a transform gate $t_i : \mathbb{R} \to \mathbb{R}$, parameterized by a weight matrix, denoted W_t^l, is represented as $t_i(\vec{h}^{l-1}) = \hat{\Phi}(W_t^l[i,:]\vec{h}^{l-1})$. Here, the gating mechanism activation function $\hat{\Phi}$ should ideally map to the open interval $(0,1)$ whereas Φ can be any non-linear activation function. The

values computed by g_i and t_i are termed as the *block state* and the *transform gate output*, respectively. Then, the block output computed by the i-th neuron is given by:

$$\vec{h}^l[i] = g_i(\vec{h}^{l-1}) \cdot t_i(\vec{h}^{l-1}) + \vec{h}^{l-1}[i] \cdot (1 - t_i(\vec{h}^{l-1})). \tag{3.16}$$

If layers $l-1$ and l differ in unit count, practitioners use techniques like zero-padding or sub-sampling. Alternatively, He et al. [He+15] used a projection matrix consisting of learnable parameters to match unit counts. We analyze what happens when gates are fully open or closed: During forward-propagation, we have

$$\vec{h}^l[i] = \begin{cases} \vec{h}^{l-1}[i] & \text{if } t_i(\vec{h}^{l-1}) = 0 \\ g_i(\vec{h}^{l-1}) & \text{if } t_i(\vec{h}^{l-1}) = 1\ . \end{cases} \tag{3.17}$$

During backward-propagation, we have

$$\frac{\partial \vec{h}^l[i]}{\partial \vec{h}^{l-1}[i]} = \begin{cases} 1 & \text{if } t_i(\vec{h}^{l-1}) = 0 \\ \frac{\partial g_i(\vec{h}^{l-1})}{\partial \vec{h}^{l-1}[i]} & \text{if } t_i(\vec{h}^{l-1}) = 1\ . \end{cases} \tag{3.18}$$

In conclusion, if the transform gate t_i outputs 0 the information from the previous layer directly passes to the subsequent layer during forward-propagation, and the global gradient directly flows from the current layer to the previous one during backward-propagation. If all gates close, the networks computes the identity function during the forward pass, and during the backwards pass a gradient highway forms carrying gradients from the output layer throughout the extent of networks of any depth to the input layer [VWB16]. In contrast, if the transform gate t_i is fully open, the gradient behaves as there was no gating mechanism. Hence, the neuron has the capacity of continuously varying between the default behavior and using the gating-mechanism.

3.4.2 *Deep Residual Networks*

Deep residual architectures are often regarded as a subset of deep highway architectures [Agg23] even though they have their own origin and unique architecture. Particularly, the degradation problem served as a motivation for designing residual networks. Moreover, in deep residual networks, unit gates are always fully open.

3.4.2.1 *The Degradation Problem*

To understand the degradation problem we consider the training error as a dependent variable and the number of neural networks layers as the independent variable. In experiments, He et al. [He+15] demonstrate that as network depth increases, accuracy first goes up, reaches a maximum, and

then starts to decrease abruptly. This drop in accuracy is clearly not due to overfitting, as these deep networks exhibit higher *training* error compared to their shallower counterparts. From a theoretical perspective, adding more layers to a model should not result in higher losses compared to a flatter model; This is because the deeper model could simply learn additional identity mappings, becoming mathematically equivalent to a shallower network. However, models struggle to learn these identity mappings even though they have the capacity to do so [He+15].

Residual connections address the degradation problem by allowing the network's input to directly pass to its output. Specifically, in networks with residual blocks, each layer connects to the next layer and also to layers $(i + r)$ steps ahead. For $r = 1$, this results in the following modification of Equation 3.16:

$$\vec{h}^{l}[i] = g_i(\vec{h}^{l-1}) + \vec{h}^{l-1}[i] \; . \tag{3.19}$$

Analogously to the deep high networks, a linear projection matrix is used in case of dimensionality mismatch between layers. The function g_i is termed the residual of the network [He+15]. Rearranging 3.19 highlights the central idea behind residual networks:

$$\vec{h}^{l}[i] - \vec{h}^{l-1}[i] = g_i(\vec{h}^{l-1}) \; . \tag{3.20}$$

Equation 3.20 demonstrates that the neural network actually learns to approximate the so-called *residual function* g_i directly, with $\vec{h}^{l-1}[i]$ added via the skip connection. If it is advantageous for the network to learn the identity mapping, the model just anneals the parameters in g_i which is more straightforward than learning an identity mapping [He+15].

3.4.2.2 *Variations and Benefits of Residual Networks*

He et al. [He+15] realized the first large scale model using residual connection, known as *ResNet*. Huang et al. [Hua+17] significantly extended this methodology in *DenseNet*. DenseNet enables each layer to utilize the feature maps of *all* preceding layers and contribute its own feature maps to all subsequent layers. Thus, if L denotes the number of layers, there are $L(L+1)/2$ residual connections. The authors' intuition behind this architecture is to facilitate the computation of complex features in later layers while allowing simple features to be passed through multiple layers until they reach a level of abstraction where they become useful [Hua+17].

Beyond mitigating the degradation problem, residual connections implicitly enable the model to search for the optimal but unknown number of hyperparameters for different tasks. Figure 3.8 visualizes this idea by unraveling a skip connection. The three skip connections correspond to eight unraveled graphs. In general n skip connections implicitly represent 2^n paths. In this context Veit et al. [VWB16] demonstrate that residual connections lead to an ensemble-like behavior, where the network learns to utilize paths of varying lengths for different tasks. Moreover, in most cases, well-trained

network for a given task favor distinct short paths dispersed throughout the architecture rather than a single long path [VWB16].

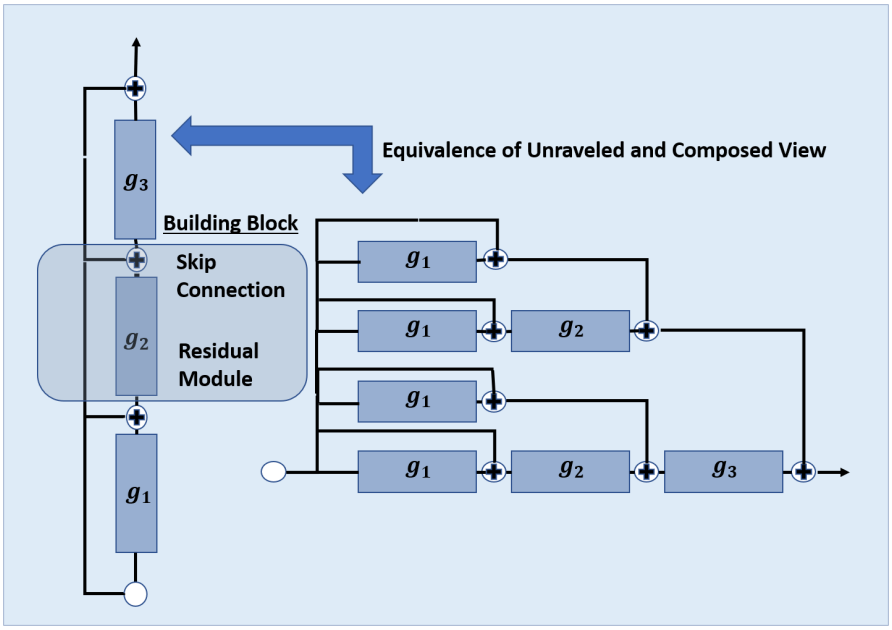

Figure 3.8: Residual Networks are shown in the left where circular nodes with a plus represent additions. Similar to Recurrent Neural Nets these networks can be unraveled as depicted on the right side. The unraveled view makes it clear that residual networks have $\mathcal{O}(2^n)$ paths connecting input and output paths.

3.4.3 *Visualizing the Benefits of Residual Connections*

In Subsection 3.3.4 Experiment 7 visualizes the stabilization effect of batch and layer normalization for an ten-layer network. Experiment 8 extends the scope to a 90-layer architecture. To remain consistent with experiment 7 we focus our investigation on early training dynamics and terminate the training process after 1,000 iterations and record the gradient distributions. We consider two models: Both consist of hundred layers where a single layer comprises a linear layer, a ReLU activation and a layer normalization module but only one model includes residual connections for *every layer*. An initial step in evaluating the training dynamics of the model involves examining the quartiles of all collected gradients for all layers:

- For the model without residual connections:
  - Lower quartile range of gradients: -1.92×10^{-7} to -6.39×10^{-9}
  - Upper quartile range of gradients: 6.62×10^{-9} to 2.26×10^{-7}
- For the model with residual connections:
  - Lower quartile range of gradients: -2.08×10^{-3} to -2.72×10^{-4}
  - Upper quartile range of gradients: 3.66×10^{-4} to 2.00×10^{-3}

In conclusion, the model without residual connections exhibits gradients that are significantly smaller in magnitude in absolute terms, indicating vanishing gradients. This observation becomes obvious when one realizes that 50 percent of the gradients are smaller than 10^{-9} in absolute terms. In contrast, the model with residual connections has gradients that are several orders of magnitude larger in absolute terms, indicating healthier gradient flow. Figure 3.9 confirms this analysis but also allows further conclusions: Without residual connections, only the later layers manifest meaningful gradient magnitudes. This suggests that the majority of the network's depth isn't substantially contributing to learning, leading to the under-utilization of model capacity. With residual connections, all layers, irrespective of depth, display significant gradient magnitudes, illustrating uniform learning across the entire depth.

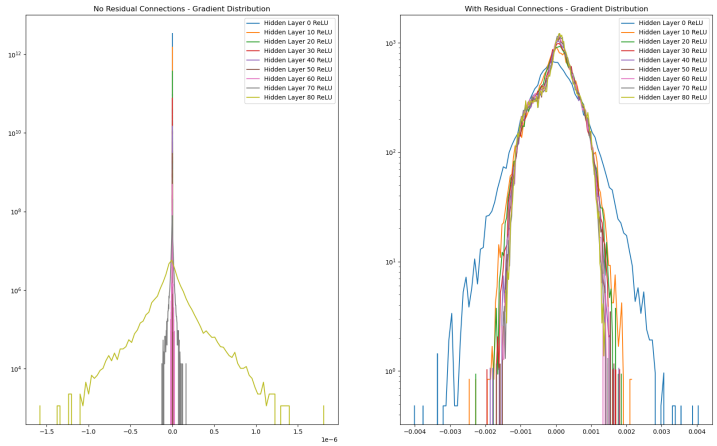

Figure 3.9: Comparison of gradient distributions between models with and without residual connections. For both subplots, the x-axis represents gradient values, while the y-axis, in logarithmic scale, indicates the frequency of these gradients. On the left, the absence of residual connections results in a concentration of gradients, on the right, with residual connections, all layers exhibit substantial gradient magnitudes.

3.4.3.1 *Conclusion*

In sum, whereas layer normalization is not sufficient for effectively training very deep neural networks, residual connections in conjunction with activation normalization solve the vanishing gradient challenge. In this vein, it is worth noting that the *scaling laws* [Hen+20] [Bah+21], established in the 2020's, state that models based on state-of-the art neural architectures follow precise power-law scaling relations, explaining the brute force approach of current AI-labs: In fact, the test loss is indirectly proportional to the data set size, the number the model's parameters and the available computing power. In essence, *future algorithmic process has become a nice bonus because it allows to*

simulate bigger models with less compute but it is not necessary for making powerful artificial brains inside computers.

4

TRANSFORMERS: OVERCOMING PARALLELIZATION AND LONG-RANGE DEPENDENCY BARRIERS

The seminal *attention is all you need* paper [Vas+17] in 2017 started a revolution in deep learning. In their paper, Vaswani et al. proposed the transformer, an architecture on which practically all recent professional large language models are built. This chapter aims to describe the architecture in detail, emphasizing its two critical properties: parallelization and the capture of long-range dependencies. This chapter starts by discussing the historical context (Section 4.1) leading to the transformer, thus providing an understanding and motivation for its elaborate architecture. Section 2.1.2 details how the transformer processes input. Then, the chapter moves to an examination of the essential components: First, we explain the basic self-attention mechanism, both as a standalone concept (Section 4.3) and how it is implemented in the transformer (Section 4.4. Section 4.5 combines all previous parts, offering a complete understanding of the transformer architecture.

4.1 HISTORICAL CONTEXT LEADING TO TRANSFORMERS

The architecture of the transformer is intricate and elaborate. Understanding its historical development illuminates how it emerged as a natural solution to the shortcomings of previous architectures. This section aims to trace this evolutionary path, culminating in the transformer, representing a significant paradigm shift due to its heavy reliance on attention mechanisms and parallelization.

The earliest efforts in language processing led to the application of the bag of words model (as detailed in Chapter 1) within single-layer and multilayer perceptron frameworks, with origins tracing back to the 1940s and 1950s [Ros58]. However, the model's inherent limitations became evident in its inability to effectively process and interpret *word order* and *long-range contextual relationships* within text. Due to their inherent ability to internally represent word order, recurrent neural networks (RNNs) dominated the research landscape in the 1960s and 1970s [MP72]. Unlike the bag of words model, which treats words as independent entities without any inherent order, RNNs process sequences of words sequentially, maintaining a hidden state that effectively captures and updates contextual information from previously processed words. The main problem with RNNs was their difficulty learning long-range dependencies within sequences due to the vanishing and exploding gradient issues during backpropagation. To resolve these limitations, Hochreiter and Schmidhuber [HS97] developed Long Short-Term Memory (LSTM) in the 1990s. LSTMs introduced specialized memory cells and gating mechanisms (as discussed in Section 3.4), allowing for better

information retention over long sequences [SVL14]. Despite these advancements, LSTMs still exhibited difficulties in handling long sequences and remained computationally inefficient due to sequential processing of tokens.

The transformer represented a radical departure from previous architectures and emerged as a groundbreaking solution to these challenges. Initially proposed by Schmidhuber in 1992 [Sch92], Vaswani et al. [Vas+17] brought this theoretical construct to life. In essence, the transformer enables *parallel processing across text sequences* while *effectively capturing long-range dependencies*. At its core lies the self-attention mechanism, initially proposed by Schmidhuber in 1993 [Sch93] and later refined by Bahdanau [BCB16]. Vaswani et al.'s main contribution was to incorporate significant algorithmic and design-related improvements of the preceding decade, such as residual connections, a modified version of the attention mechanism, and layer normalization, uniquely and efficiently. Moreover, their paper came with an effective choice of hyperparameters that have remained broadly consistent. Even though the transformer architecture is criticized for its tendency to require substantial amounts of training data and computational resources, prominent figures such as Ilya Sutskver recently argued that this architecture can *obviously lead to artificial general intelligence* [Sut23].

4.2 INPUT PROCESSING

In its purest form, the transformer can perform any set-to-set tasks where one focuses solely on the relationships and interactions between elements of the input set, making it an extremely general architecture. However, in its initial design, the transformer is supposed to handle any form of data with sequential semantics such as text, time series [Zho+21], or genomics data [CL23]. More formally, let V denote a predefined vocabulary, encompassing all possible tokens encoded as integers. Then, the transformer processes input sequences s of the form: $s = \{w_1, w_2, \ldots, w_m\}$, where each token w_i belongs to the vocabulary V. Initially, we must ensure that the input length is smaller than the sequence length the transformer can handle. This maximum sequence length denoted l_{max} forms a hyperparameter of the architecture. To deal with sequence lengths unequal to l_{max} we can use to techniques:

- **Chunking**: For sequences where $m > l_{max}$, we partition the input sequence s into non-overlapping contiguous chunks. Each chunk is denoted by s_i and has a length l_i that satisfies $l_i \leqslant l_{max}$. These chunks are successively processed.
- **Padding**: Conversely, for sequences shorter than l_{max}, the transformer extends the sequence length by appending zero entries until it matches l_{max}.

Next, we map all tokens to embeddings as discussed in Subsection 2.1.3.1. Algorithm 4.1 reproduces the approach for a single token.

Up to this point, the transformer behaves like a bag of words model, lacking an understanding of sequence or spatial relationships among tokens.

Listing 4.1: Token Embedding

```
class TokenEmbedding(object):
    Input: w_i ∈ V, a token ID.
    Output: e⃗_e ∈ ℝ^{d_in}
    Parameters: W_e ∈ ℝ^{d_in × |V|}
        return e⃗_e = W_e[:, w_i]
```

Vaswani et al. introduce positional encodings to leverage the local structure in the data. We discuss two prevalent approaches for positional encoding. Let l be the position of the token we want to embed:

- **Learnable Embeddings**: Similar to how we embed token IDs, we can also embed positions. Specifically, we introduce a new matrix $W_p \in \mathbb{R}^{d_{in} \times l_{max}}$ that maps an input position l within the range $[1, 2, \ldots, l_{max}]$ to its vector representation $\vec{e}_p \in \mathbb{R}^{d_{in}}$. A limitation of this approach is its inability to generalize to sequence positions unseen during training, particularly for $l \geqslant l_{max} + 1$.

- **Sinusoidal Encoding**: Vaswani et al. [Vas+17] propose sinusoidal encodings for handling sequences of arbitrary length. As before we introduce a new matrix $W_p \in \mathbb{R}^{d_{in} \times l_{max}}$ and define its l-th column:

$$W_p[2i-1, l] = \sin\left(\frac{l}{\nu^{2i/d_e}}\right), \tag{4.1}$$

$$W_p[2i, l] = \cos\left(\frac{l}{\nu^{2i/d_e}}\right), \tag{4.2}$$

  where $0 < i \leqslant d_e/2$ and ν is an additional hyperparameter fixed to $\nu := 10,000$.

The hyperparameter ν essentially modulates the frequency range of the sine and cosine functions. To visualize this, I experimentally (experiment 9) varied this hyperparameter and displayed the results in Figure 4.1. First, sinusoidal functions have a clear, repeating pattern that helps the model learn these positional relationships. Moreover, with a base of 10,000, the frequency of the sinusoidal waves in the positional encoding is much lower, meaning the encoding changes more slowly as one moves along the position axis. This implies a smoother gradient of positional information, which could help when the model needs to generalize across positions that were not explicitly seen during training. Conversely, with a base of 10, the frequency is higher, which means the model will have a more distinct positional signal for closely situated tokens. While this can enhance the model's ability to distinguish between positions, it might also cause the model to overfit to the position-specific patterns found in the training data, reducing its ability to generalize to unseen positions or to larger contexts where such fine-grained positional distinctions are not as relevant or are different.

The positional embedding for the token and the token embedding are usually summed up. More precise, for a token w_l of a sequence of tokens, the embedding is defined as

$$\vec{e} = W_e[:, w_l] + W_p[:, l] \ .$$

The intuition here is that adding the position embeddings to the semantic embeddings provides interpretable distances between the embedding vectors that the model can learn.

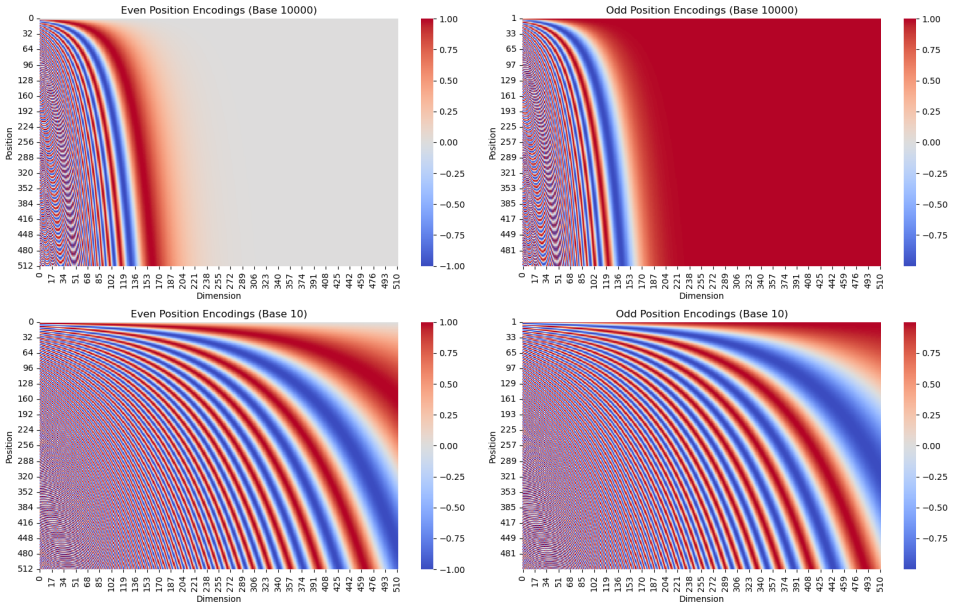

Figure 4.1: Comparison of positional encodings generated using two different bases: 10000 (top row) and 10 (bottom row). In these heatmaps, the x-axis represents the dimensions (from 0 to 512), and the y-axis represents the positions (even and odd positions are segregated, ranging from 0 to 512 for even and 1 to 511 for odd). The color gradient in the heatmap indicates the magnitude of the positional encoding values. A transition from blue to red corresponds to a change from the minimum to the maximum encoding value for that specific position and dimension.

4.3 BASIC SELF-ATTENTION

As highlighted in Section 4.1, the cornerstone of the transformer architecture is self-attention because this mechanism enables the model to learn long-range dependencies. While many textbooks introduce this concept through mathematical formulas, they often overlook the intuitive underpinnings of self-attention [Agg18] [Agg23] [GBC18]. This section aims to provide a coherent and accessible motivation for this concept.

4.3.1 *Basic Concept of Self-Attention*

Human cognitive processes inspire the self-attention mechanism. When humans process information, they do not consider all stimuli equally. Instead, they focus on specific parts based on their relevance to the task. For instance, when reading a text, the brain pays more attention to crucial words for understanding the main idea while glossing over less essential words. This selective focus is often based on how relevant or similar these parts are to the object of focus. The self-attention mechanism in transformers is designed to mimic this ability [BCB16].

Mathematically, this mechanism is represented as a sequence-to-sequence function. Given an input sequence of real-valued vectors $(\vec{e}_1, \vec{e}_2, \cdots, \vec{e}_t)$ where each vector has dimension $\mathbb{R}^{d_{in}}$, self-attention computes a sequence of real-valued output vectors $(\vec{y}_1, \vec{y}_2, \cdots, \vec{y}_t)$ of dimensionality $\mathbb{R}^{d_{out}}$. In this context, the *self* in self-attention signifies that the mechanism computes attention scores concerning the elements of the input sequence itself rather than external data. Each output vector $\vec{y}_i$ represents the context-enriched representation of the i-th token in the sequence computed as follows:

$$\vec{y}_i = \sum_{j=1}^{t} w_{i,j} \vec{e}_j \quad \text{subject to} \quad \sum_{j=1}^{t} w_{i,j} = 1 \quad \forall 1 \leqslant i \leqslant t \tag{4.3}$$

A weight, denoted as $w_{i,j}$, represents how much attention the i-th token should pay to token $\vec{e}_j$ in the sequence. These weights are determined based on a similarity measure detailed in Subsection 4.3.2. In consequence, $\vec{y}_i$ encapsulates not only the features of the i-th token but also the entire context. In other words, each element in the sequence pays attention to all other elements in the same sequence regardless of their distance, enabling the model to capture long-range dependencies that RNNs struggle with.

To understand the practical application of this mechanism, consider a pizzeria shop owner who wants to recommend pizzas to customers. The shop owner has a list of features for each pizza, such as *vegetarian*, *meat*, *fish*, and *cheese*. Similarly, each customer has preferences like *vegetarian*, *meat-fan*, *fish-fan*, and *cheese-fan*. The task is to match each customer's preferences with the features of each pizza to make a personalized recommendation. Here, the attention mechanism would involve computing a weight for each pizza based on how well its features align with a customer's preferences. The more aligned they are, the more attention that pizza gets for that particular customer, guiding the recommendation process.

4.3.2 *Measuring Similarity*

One commonly used mathematical operation to measure similarity is the dot product. In the case of normalized vectors, the dot product computes the cosine similarity and essentially measures the cosine of the angle between the vectors, capturing the geometric relationship between them. However, the

dot product is computationally less expensive compared to cosine similarity, which requires additional operations for magnitude normalization.

Given two vectors $\vec{e}_i$ and $\vec{e}_j$, their dot product $\vec{e}_i \cdot \vec{e}_j$ serves as the raw weight, denoted $\hat{w}_{i,j}$. We normalize the raw weights to ensure that they are comparable across different vectors. For this normalization, the transformer applies the softmax function. In essence, the softmax function transforms raw weights into a probability distribution, quantifying the likelihood of each token in a sequence being the focal point of attention. Thus, the formula to compute a normalized weight $w_{i,j}$ is:

$$w_{i,j} = \frac{\exp \hat{w}_{i,j}}{\sum_{m=1}^{t} \exp \hat{w}_{i,m}} \,. \tag{4.4}$$

Note that each token in a sequence is represented by a high-dimensional embedding vector which are optimized during the training process to capture the semantic meaning. Thus, vectors representing semantically similar concepts are near each other minimizing the angle between them and thus maximizing the cosine of this angle leading to a high dot product. Continuing the pizzeria example, each pizza and customer is represented by a feature vector that encapsulates various attributes like *vegetarian*, *meaty*, *marine*, or *cheesy* for pizzas and corresponding preferences for customers. The central idea here is that if two tokens are semantically similar or contextually relevant, their embedding vectors will be closely aligned in the high-dimensional embedding space resulting in a high value of similarity. We conclude this section with two observations:

Observation 2 (Attention and Space) It is worth noting that the self-attention mechanism itself does not inherently model the concept of sequence order or space. In the high-dimensional embedding space, attention views the input sequence essentially as a bag of words. This means that if the input sequence is permuted, the output sequence would also be permuted in the same way, but the values themselves would remain unchanged. Conversely, this property allows the model to naturally model long-term dependencies within a sequence.

Observation 3 (Soft Attention) An important requirement for any component of a neural network architecture is differentiability to facilitate backpropagation. In the case of the attention mechanism, we utilize *soft attention*, which employs the softmax function to assign attention weights across all input tokens in a differentiable manner, thus enabling the network to adaptively learn the significance of different sequence parts during training.

4.4 SCALED SELF-ATTENTION IN THE TRANSFORMER

Building on the basic self-attention mechanism outlined in Section 4.3, we discuss how this concept is applied in the transformer. There are three important modifications: first, the infusion of learnability into self-attention via

additional parameters, as detailed in Subsection 4.4.1; second, the implementation of parallel processing to handle entire token sequences simultaneously, as elucidated in Subsection 4.4.2; third, scaling the input prior to the softmax application to ensure stability during gradient descent, as discussed in Subsection 4.4.3.

4.4.1 *Learnable Self-Attention*

When revisiting Equations 4.3 and 4.4 we notice that every vector $\vec{e}_i \in \mathbb{R}^{d_{in}}$ entering the attention layer will influence the attention over the whole sequence in three distinct ways:

1. The dot product between $\vec{e}_i$ and every other vector is computed resulting in the raw weights $\hat{w}_{i,j}$. This role is the *query role* of $\vec{e}_i$.
2. Every other vector $\vec{e}_j$ in the sequence including $\vec{e}_i$ itself, will use $\vec{e}_i$ to establish its own raw weights $\hat{w}_{j,i}$. This role is the *key role* of $\vec{e}_i$.
3. Given all weights of a sequence, $\vec{e}_i$ takes part in the computation of each output vector $\vec{y}$. This is the *value role* of $\vec{e}_i$.

To accommodate these three roles for every single vector $\vec{e}_i$ Vaswani et al. [Vas+17] introduce additional parameters [Vas+17]. These parameters give the attention layer the ability to control and modify the incoming vectors to suit the three roles they must fill. Mathematically, we introduce separate matrices W_q, W_k, W_v for transforming the input vectors into query, key, and value representations. Algorithm 4.2 outlines their use for bidirectional self-attention, a variant of the self-attention mechanism that considers the entire context, that is both past and future tokens, when computing the attention for each individual token: On every input vector $\vec{e}_i \in \mathbb{R}^{d_{in}}$ we apply a transformation to compute three vectors: the *query vector* $\vec{q}_i \in \mathbb{R}^{d_{attn}}$, the *key vector* $\vec{k}_i \in \mathbb{R}^{d_{attn}}$ and the *value vector* $\vec{v}_i \in \mathbb{R}^{d_{out}}$ corresponding to their different roles in the attention layer. After these transformations, the rest of the attention mechanism follows the computations for the basic self-attention mechanism detailed in Section 4.3. Bidirectional self-attention is applicable in neural language translation tasks, as it allows to attend to all tokens in the source language to predict the subsequent token in the target language, thereby capturing complex dependencies.

4.4.2 *Efficient Auto-regressive Self-Attention*

Algorithm 4.2 processes a single token. To exploit the efficiency of modern hardware in matrix multiplication, we stack the vectors representing the tokens in a single $d_{in} \times l_x$ matrix, denoted X, where l_x corresponds to the number of tokens in the current sequence. This approach enables parallel processing of all tokens in an entire sequence, a key advantage over RNN-based architectures where tokens of a sequence are processed sequentially. Large language models such as GPT-n are autogressive models meaning

Listing 4.2: Bidirectional Self-Attention

$\vec{v} \leftarrow$ `Bidirectional Self-Attention` $(\vec{e_i}, (\vec{e}_1, \cdots, \vec{e}_{l_x}), W_q, W_k, W_v)$:

```
    Input:
        e_i, the vector representation of the token currently being
            predicted.
    Input:
        ∀j : e_j, vector represention of context tokens.
    Output:
        y_i ∈ R^{d_out} vector representation of the combined token & context
            .
    Parameters:
        W_q, W_k ∈ R^{d_attn × d_in} , W_v ∈ R^{d_out × d_in} .
    Computation:
        q_i ← W_q e_i
        ∀j : k_j ← W_k e_j
        ∀j : v_j ← W_v e_j
        ∀j : w_{i,j} = exp(q_i · k_j)/√d_attn / Σ_{m=1}^{t} exp(q_i · k_m)/√d_attn
        return y_i = Σ_{j=1}^{t} w_{i,j} v_j
```

that they aim to learn a probability distribution over a single token conditioned on its preceding tokens. Therefore, given the token currently being predicted, unlike bidirectional self-attention models, autoregressive models prevent future tokens from being accessed during the attention phase. To enforce this, we introduce a $l_x \times l_x$ binary masking matrix where the element $Mask[t_z, t_x]$ controls the visibility of token t_x to token t_z. In the case of autoregressive models we set this matrix to:

$$Mask[t_z, t_x] = \begin{cases} 1, & \text{if } t_z \leqslant t_x \\ 0, & \text{otherwise} \end{cases} \tag{4.5}$$

where t_z and t_x are token indices in X. To further simplify the notation, we define a matrix to matrix softmax function $S : \mathbb{R}^{l_x} \times \mathbb{R}^{l_x} \to \mathbb{R}^{l_x} \times \mathbb{R}^{l_x}$ [PH22]. Given a matrix A, its entry at position $[t_z, t_x]$ evaluates to:

$$S(A)[t_z, t_x] := \frac{\exp(A[t_z, t_x])}{\sum_{t'=1}^{l_x} \exp(A[t', t_x])}, \tag{4.6}$$

where the summation is done over all $1 \leqslant t' \leqslant l_x$. With these notational conventions, Algorithm 4.3 handles entire input sequences, while also incorporating a masking mechanism necessary for autoregressive language models. In Algorithm 4.3 we calculate a score matrix S which quantifies the relevance or affinity between each query and each key in the sequence. Note that we set an entry to $-\infty$ if the corresponding entry in the mask is 0. After applying the softmax this entry evaluates to 0 rendering it invisible for token prediction. Moreover, the score matrix S is scaled by factor $\sqrt{d_{attn}}$

as explained in 4.4.3. In the case of bidirectional self-attention, we invoke Algorithm 4.3 passing a mask defined by $Mask[t_z, t_x] = 1$ for all entries.

Listing 4.3: Auto-Regressive Self-Attention

$\hat{V} \leftarrow$ `Auto-Regressive Self-Attention` $(X, W_q, W_k, W_v, Mask)$:
    `Input:`
        $X \in \mathbb{R}^{d_e \times l_x}$, `the vector representations of the sequence.`
    `Output:`
        $\hat{V} \in \mathbb{R}^{d_{out} \times l_x}$ `representation of tokens of` X `combined with context information.`
    `Parameters:`
        $W_q \in \mathbb{R}^{d_{attn} \times d_e}$,
        $W_k \in \mathbb{R}^{d_{attn} \times d_e}$,
        $W_v \in \mathbb{R}^{d_{out} \times d_e}$, .
    `Hyperparameters:`
        $Mask \in \{0,1\}^{l_x \times l_x}$.
    `Computation:`
        $Q \leftarrow W_q X$ `[Query` $\in \mathbb{R}^{d_{attn} \times l_x}$`]`
        $K \leftarrow W_k X$ `[Key` $\in \mathbb{R}^{d_{attn} \times l_x}$`]`
        $V \leftarrow W_v X$ `[Value` $\in \mathbb{R}^{d_{out} \times l_x}$`]`
        $S \leftarrow K^T Q$ `[Score` $\in \mathbb{R}^{l_x \times l_x}$`]`
        $\forall t_z, t_x,$ `if` $\neg Mask[t_z, t_x]$ `then` $S[t_z, t_x] \leftarrow -\infty$

        `return` $\hat{V} = V \cdot softmax(S/\sqrt{d_{attn}})$

To visualize how the transformer architecture optimizes computational efficiency by processing all tokens in a sequence in parallel, we refer to Figure 4.2. The figure illustrates a dual-directional approach to token processing. Vertically, each token ("She", "loves", "Salami", "Pizza") is transformed in parallel through matrices W_Q, W_K, and W_V, highlighting computational efficiency. Horizontally, after transformation, tokens exchange contextual information with all other tokens in the sequence *independently of their distance* to the token currently being predicted. This parallelization not only improves computational speed but also captures relationships among tokens within the sequence. By efficiently processing all tokens in parallel, the transformer outperforms LSTMs that process tokens sequentially and have problems with tokens that are not in the immediate neighborhood of the token currently being predicted.

4.4.3 *Scaled Attention*

The second modification to the basic attention mechanism introduced by Vaswani et al. [Vas+17] addresses the sensitivity of the softmax function to large input values during backpropagation. Specifically, it proposes a rescaling of the inputs to mitigate this issue. To make this precise, consider a scal-

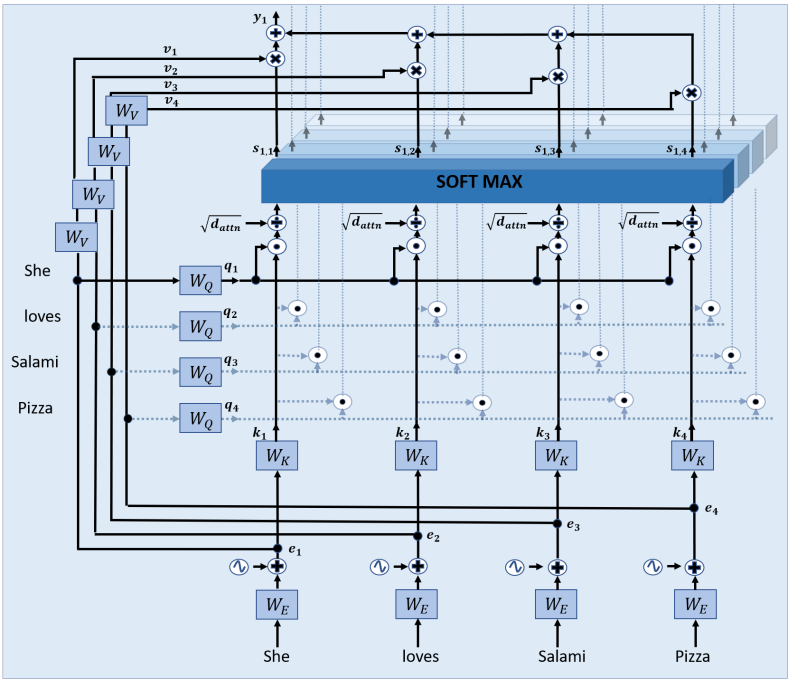

Figure 4.2: The visualization of self-attention in the transform as a circuit demonstrates the parallel efficiency advantages of this mechanism

ing factor $\kappa \in \mathbb{R}$ and inputs $z_1, \cdots, z_s$ to the softmax function. For distinct indices i and j, we have:

$$\frac{\frac{\exp(\kappa z_i)}{\sum_{m=1}^{s} \exp(\kappa z_m)}}{\frac{\exp(\kappa z_j)}{\sum_{m=1}^{s} \exp(\kappa z_m)}} = \left(\frac{\exp z_i}{\exp z_j}\right)^{\kappa}$$

Further, as κ scales towards infinity, the expression simplifies to:

$$\lim_{\kappa \to \infty} \left(\frac{\exp z_i}{\exp z_j}\right)^{\kappa} = \begin{cases} \infty & \text{if } z_i > z_j \\ 1 & \text{if } z_i = z_j \\ 0 & \text{if } z_i < z_j \end{cases} \tag{4.7}$$

We repeat our observation from Subsection 3.2.4.1 that scaling the inputs to the softmax causes the largest value to approach 1 exponentially fast, while all smaller inputs approach 0 also exponentially fast. Based on Equation 4.7, we study the effect of input scaling on the gradients of the softmax function. For convenience, we reintroduce the softmax function as a vector-to-vector mapping $\text{softmax} : \mathbb{R}^d \to \mathbb{R}^d$, and its gradient is represented by the Jacobian matrix $J \in \mathbb{R}^{d \times d}$:

$$J[i, j] = \frac{\partial s_i}{\partial z_j} = s_i \cdot (\delta_{ij} - s_j) \text{ for all: } i, j \in [d_e] \, .$$

For ease of understanding, consider $d = 4$ and write down the Jacobian matrix explicitly:

$$J = \begin{pmatrix} s_1(1-s_1) & -s_1s_2 & -s_1s_3 & -s_1s_4 \\ -s_2s_1 & s_2(1-s_2) & -s_2s_3 & -s_2s_4 \\ -s_3s_1 & -s_3s_2 & s_3(1-s_3) & -s_3s_4 \\ -s_4s_1 & -s_4s_2 & -s_4s_3 & s_4(1-s_4) \end{pmatrix}$$

As the softmax input is scaled, the output s_i nears either 0 or 1. The impact on the Jacobian matrix J is:

- Off-Diagonal Elements $-s_is_j$ (for $i \neq j$): When the input to the softmax is scaled, the maximum softmax output s_i will approach 1, while all other s_j where $j \neq i$ will approach 0. As a result, for each pair (i, j), at least one of s_i or s_j will be close to zero, making the product $-s_is_j$ near zero.

- The diagonal elements $s_i(1-s_i)$ also approach zero as s_i nears 0 or 1.

A Jacobian matrix populated primarily with small values implies that the local gradient, when backpropagated, will also be small. During backpropagation, this local gradient is multiplied with the backpropagated global gradient, leading to the vanishing gradient problem and impeding the training process. To alleviate the vanishing gradient problem, we must ensure that the input to the softmax remains at a reasonable scale. In this vein, let us consider how the input to the softmax function scales with its dimensionality d. We present two distinct but complementary ideas that motivate the need for rescaling the inputs to the softmax function:

1. Geometric Perspective: The first idea is rooted in the geometric properties of vectors in high-dimensional spaces. For instance, take a vector $\vec{v} = (c, c, \ldots, c)^T \in \mathbb{R}^d$. The L_2 norm of this vector is $\|\vec{v}\|_2 = \sqrt{d} \cdot c$. We observe that the L_2 norm of a vector inherently scales with the square root of its dimensionality d.

2. Statistical Perspective: The second idea provides a statistical rationale, noting that if the components of key and query vectors $\vec{q} \in \mathbb{R}^d$ and $\vec{k} \in \mathbb{R}^d$ (whose dot product is normalized by the softmax function) are independent random variables with a mean of 0 and a variance of 1, the variance of their dot product will be d as explained in Lemma 2.

Vaswani et al. [Vas+17] provide a pragmatic solution by simply scaling the input to the softmax:

$$\frac{q^T k}{\sqrt{d}}.$$

In conclusion, this rescaling effectively counteracts the vanishing gradient problem, making the training process more stable.

4.5 COMPOSING THE TRANSFORMER

After exploring the individual components of the transformer, we now transition to an analysis of how Vaswani et al. [Vas+17] assembled these elements to construct the transformer architecture. Subsection 4.5.1 delves into the implementation of self-attention within the transformer framework. This is followed by an examination of a singular transformer block in Section 4.5.2. In Section 4.5.3, we integrate these insights to present an overview of three principal transformer variants: the encoder-only, decoder-only, and the encoder-decoder models.

4.5.1 *Multiple Attention Heads*

In the transformer, self-attention is applied with several self-attention mechanisms in parallel. For every separate attention mechanism $h \in [1, \cdots, H]$, and $H \in \mathbb{N}$, called attention head, we introduce distinct matrices W_q^h, W_k^h, W_v^h. For input $\vec{x}$ each attention head computes an output $\vec{y}^h$. These vectors are concatenated to a single output vector $\vec{y}$. On this output vector an additional linear transformation is applied. Algorithm 4.4 depicts this process.

To maintain efficiency while taking advantage of multiple parallel attention mechanisms, each head calculates low-dimensional key queries and values. For example, if the input vector is of dimension $k = 128$ and we have $h = 4$ attention heads, the input vectors are projected by 32×128 transformation matrices. This requires $3h$ matrices (a key, query, value matrix for every head) of size $k \times \frac{k}{h}$. This yields $3hk\frac{k}{h} = 3k^2$ parameters to compute the inputs to the multi-head self-attention which is roughly the same as the parameters for the single-head attention mechanism.

Remark 7 (Intuition behind Multi-Head Attention) Vaswani et al. argued that multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions of the sequence. However, Cordonnier et al. [CLJ20] observed that many attention heads learn redundant key-query projections countering this view. Furthermore, Schlag et al. [Sch+19] found empirical evidence that an individual attention head does not access merely a subset of the information in the attended token vector but instead captures nearly the full information content. To my knowledge, there is no clear understanding why multi-head attention works well. Hence, next to the proposed attention mechanisms by Schlag et al. [Sch+19] several competing attention mechanisms have since been discussed in the literature [CLJ20].

4.5.2 *The Transformer Block*

The Transformer architecture is built around the self-attention mechanism and designed for modularity. The modularity is achieved by the stacking of multiple identical blocks, each consisting of two primary components: the

Listing 4.4: Basic-Single Query in Pseudo-Code

```
MultiHeadAttention(X, H, (W_q^h, W_k^h W_v^h), W_o, Mask):
    Input:
        X ∈ ℝ^{d_e × l_x}, the vector representations of the token sequence.
    Output:
        V̂ ∈ ℝ^{d_out × l_x} representation of tokens of X combined with context
             information.
    Parameters:
        For h ∈ [1, ··· H]:
                W_q^h ∈ ℝ^{d_attn × d_e},
                W_k^h ∈ ℝ^{d_attn × d_e},
                W_v^h ∈ ℝ^{d_mid × d_e}.
        W_o ∈ ℝ^{d_out × H·d_mid}.
    Hyperparameters:
        Mask ∈ {0,1}^{l_x × l_x},
        H, the number of attention heads.
    Computation:
        For h ∈ [1, ··· H] do:
            Y^h ← Auto-Regressive Self-Attention (X, W_q^h, W_k^h, W_v^h, Mask)
        Y ← Y^1 ∘ Y^2 ∘ ··· ∘ Y^H

        return V̂ = W_o Y
```

multi-head self-attention layer and a position-wise feed-forward network. Every block starts with the masked multi-head self-attention layer where the tokens exchange context information. The output, enriched with contextual dependencies, is then directed to the feed-forward layer. This layer consists of two linear transformations with ReLU activation in between. Specifically, every token is passed in parallel through its own feed-forward layer, enabling the network to process the context-enriched versions of the original tokens independently from the other tokens in the same sequence. Both the multi-head self-attention and the feed-forward network are equipped with residual connections, followed by layer normalization. Different implementations introduce different orders of the components. Note however that the feedfoward-network, the self-attention, the layer normalization and the residual connections are conserved throughout all different variants of the transformer architecture, and thus constitute the essential components of the transformer [Xio+20]. The only significant modification in the original transformer has been the recommendation to apply layer normalization before attention and linear layers because this simple change greatly stabilizes the gradients at initialization and reduces training time [Xio+20]. Recently, He and Hoffmann [HH23] proposed a radically simplified transformer block where they demonstrated the feasibility of removing the matrices W_v^h and W_o^h in the multi-head attention and even of removing layer normalization and residual connections.

4.5.3 *Stacking Blocks and Transformer Variants*

The Transformer model is constructed by stacking a series of these blocks, typically ranging from 6 to 12 depending on the model size. The input to the first block is the sequence of embeddings, which are enhanced with positional encoding to retain positional information. Each subsequent block receives the output of the previous block as its input. In this stacked architecture, each block refines the representations from the previous block, gradually building more abstract and contextually rich representations of the input sequence. The only interaction between tokens is through self-attention which constitutes a defining property of the transformer. The final block's output is then typically used for task-specific purposes. For instance, in language prediction, the output is passed to a linear layer followed by a softmax to produce a probability distribution over the target vocabulary for each position in the output sequence.

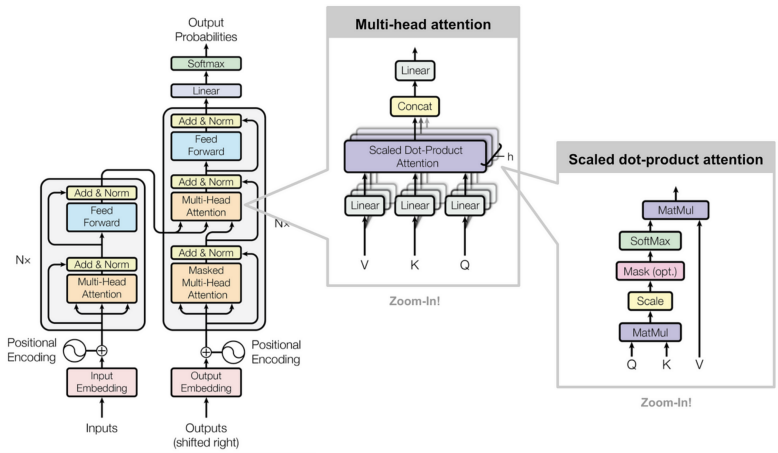

Figure 4.3: The depicted architecture on the left is partitioned into two sections: the left portion illustrates the architecture of the encoder, while the right portion delineates the structure of the decoder. This design allows for flexible configurations. Depending on requirements, one can construct a model employing only the encoder, exclusively the decoder, or an integrated encoder-decoder architecture. Note that in the here-depicted encoder-decoder version, the decoder block includes an additional multi-head attention layer that attends to the output of the encoder. Adapted from [Ala18].

As a side-node, many textbooks present a more specialized encoder-decoder transformer architecture for neural language translation originally introduced by Vaswani et al. [Vas+17]. I described here the decoder-only variant of the transformer, the underlying architecture for today's large language models such as Chat-GPT and Google's BARD [BK23]. A representative of the encoder-only architecture is Google's BERT [Dev+19]. Figure 4.3 depicts all three variants.

5

A NOVEL METHOD FOR AUTOMATED NEURAL NETWORK INTERPRETABILITY IN BIOLOGICAL APPLICATIONS

Initially, deep neural networks excelled in traditional machine learning tasks such as image recognition and natural language processing. Marking a paradigm shift, AlphaFold, developed by DeepMind, revolutionized protein structure prediction through its AI-driven deep learning system, achieving unprecedented accuracy in the *Critical Assessment of Structure Prediction* (CASP) competitions and thus demonstrating the potential of neural networks in biological and medical research [Jum+21]. Subsequently, *neural network interpretability* gained prominence because understanding the internal workings of neural networks not only increases their trustworthiness in real-world applications but could also lead to breakthroughs in any research area that heavily relies on vast quantities of data, such as genomics, genetics, medical imaging, and drug discovery. For instance, the information stored in the weights and activations used by a convolutional neural network to automatically detect gastric cancer in endoscopic images [Hir+18] could provide insights into the distinguishing features of cancerous versus non-cancerous tissue. Making these features human-interpretable could potentially reveal biomarkers or new understandings of the disease's progression that can inform more targeted treatment approaches.

Furthermore, this research contributes to the discussion on the fundamental problem of *superalignment*, in which human evaluators seek to direct and evaluate the performance of a superhuman model, to determine if its outputs correspond with human intentions, even in cases where humans may not fully grasp these outputs [Bur+23]. For instance, such superhuman models could try to deceive humans by mimicking human reasoning patterns. In this vein, models could present arguments or solutions that superficially align with human logic but are subtly skewed to achieve the AI's objectives. This could involve using familiar reasoning styles to gain trust while inserting nuanced manipulations in the logic that are hard for humans to detect. Note that this idea was generated using ChatGPT-4, which was asked to create scenarios on how an AI could deceive humans. If scaled, the proposed method is designed to potentially uncover the internal mechanisms of an AI and render its concealed intentions transparent.

Section 5.1 outlines the two principal motivations for this novel method for automated neural network interpretability: First, an overview of traditional interpretability methods is provided, highlighting their current abilities and limitations. The discussion underscores the need for methodologies that provide practitioners with more comprehensive and human-interpretable insights. The second motivation emerges from the unprecedented success of

large language models that combine unsupervised and supervised learning to build creative and general intelligence systems capable of human-level reasoning in many areas [Bub+23]. I contribute by proposing a method that transfers this paradigm, also called semi-supervised learning or weak supervision, to the field of neural network interpretability.

Section 5.2 outlines the technical details of the experimental setup and discusses the results in the context of this proof-of-work study. Section 5.3 is dedicated to analyzing the significance and limitations of this method in relation to existing literature and proposes several avenues of research to generalize and extend this method.

5.1 RELATED WORK AND NEW CONTRIBUTION

This section begins by outlining the challenges and limitations inherent in current interpretability techniques, particularly in the context of biological data analysis (see Subsection 5.1.1). Building upon this foundation, I delve into the potential of unsupervised learning for grasping abstract concepts without direct instruction and discuss its successful integration with supervised learning to refine models for specific tasks (see Subsection 5.1.2). Additionally, I present a study that has successfully applied unsupervised learning to representations of flattened images, which is pertinent to our proposed method that interprets flattened and concatenated filter activation vectors during training. Subsections 5.1.1 and 5.1.2 lay the groundwork for the novel approach, wherein a pre-trained neural network interpreter is further trained to decipher the internal mechanisms of a target neural network's activations. Subsection 5.1.3 provides a broad overview of this novel method.

5.1.1 *Interpretation Approaches and their Challenges and Limitation*

Methods for understanding, explaining and visualization of the internal and overall behavior of deep neural networks are broadly categorized into *visualization techniques*, *attribution methods*, and *surrogate modeling* [Sha21]. Visualization techniques, such as feature and saliency maps, aim to highlight regions in the input data that significantly influence the model's output [Sha21]. Attribution methods like *layer-wise relevance propagation* [Mon+19] and *DeepLIFT* [SGK17] assign importance scores to input features, attempting to explain model predictions [Sha21]. Surrogate models involve constructing simpler, more interpretable models that approximate the predictions of neural networks [MKM19].

Despite these developments, there remains a gap in the ability to interpret the behavior of neural networks as a whole. Existing methods often struggle to provide a comprehensive understanding of how deep and complex architectures transform and analyze input data. This limitation is particularly evident in the analysis of biological data, where the interpretation requires not just a technical explanation but also a meaningful translation into biological insights. The ideal interpretability technique would involve a method capa-

ble of examining the intricate activations within the neural network. This method would then articulate, in human language, the reasons behind the network's specific decisions. Additionally, it would describe the biological insights encoded in the features calculated by the network, elucidating their meaning and how they influence the decision-making process of the neural network.

5.1.2 *Unsupervised Learning and Supervised Learning*

This section presents empirical evidence on how neural networks can internalize complex, abstract concepts through unsupervised learning. In this vein, the literature discussed also emphasizes the potential of fine-tuning pre-trained models with a minimal supervised data set so that these models can perform various tasks.

5.1.2.1 *The Magic of Unsupervised Learning*

Radford et al.'s seminal study [RJS17] demonstrates the neural network's capacity to internalize abstract concepts through unsupervised learning *when provided with enough computational power and diverse training data*. In detail, the researchers trained a recurrent neural network on the Amazon product review data set [MPL15] in an unsupervised manner. They discovered that this network learned disentangled, high-level representations implicitly present in the training data. For instance, a neuron was identified that intrinsically performed sentiment analysis, an ability that was *not explicitly taught*. The unit's capability was benchmarked on the Stanford sentiment treebank, where it attained state-of-the-art results, underscoring the efficacy of unsupervised learning in developing meaningful features from raw data [RJS17].

Importantly, Radford et al.'s study also indicates that models, pre-trained on large and diverse data sets, can be effectively fine-tuned for various tasks like sentiment analysis with *minimal labeled data*. This combined approach of extensive unsupervised and minimal supervised learning matches the performance of strong baseline models trained on extensive supervised data sets. One could argue that this approach holds particular promise in fields such as biology, where vast amounts of data exist, but labeled examples are limited or difficult to obtain.

GPT-2, with its 1.5 billion parameters, exemplifies a significant iteration of this methodology [Rad+19]. First, GPT-2 underwent an expansive unsupervised learning phase on the 40 GB *WebText* data set consisting of scraped text reached through outbound links from Reddit [Rad+19]. In a second phase, the model was fine-tuned to perform diverse tasks such as text summarization, reading comprehension, and translation on a much smaller supervised data set. A result of this process is the model's ability to generate coherent answers to questions on the *conversational questing answering challenge* (CoQA) data set. This challenge measures the ability of machines to understand a text passage and answer a series of interconnected questions that appear in a conversation. Crucially, this performance, which matches or ex-

ceeds that of several baseline systems, is attained despite GPT-2 not being explicitly trained on the over 127,000 training examples in the CoQA data set. This underscores GPT-2's capacity to leverage the broad spectrum of language patterns it has absorbed from the vast, unstructured real-world data in WebText.

In sum, I have examined empirical evidence that complex features like sentiment are learned as *incidental features* through extensive language understanding. This line of research demonstrates how unsupervised training contributes to learning abstract concepts. Furthermore, the literature suggests that models, once pretrained, can be fine-tuned with a small number of labeled examples to achieve high performance on various tasks.

5.1.2.2 *Unsupervised Image Learning Without Spatial Information*

In the context of unsupervised representation learning for images, my approach also draws empirical support from a study conducted by Mark Chen et al. [Che+20]. Their research demonstrates that sequence transformer models, similar in scale to GPT-2, can effectively learn to predict image pixels in a *flattened vector format*, without any prior knowledge of the images' two-dimensional structure. Despite the absence of spatial awareness and positional encodings, the model developed robust image representations. When assessed through linear probing on the CIFAR-10 data set, the model achieved an accuracy of 96.3 percent, comparable to leading supervised pretrained models. This research is relevant to this study, where image vectors and filter activations are flattened.

5.1.3 *A Novel Method For Automated Interpretability*

I give a brief non-technical overview of the novel method for automated interpretability. Afterward, Section 5.2 describes the approach in more detail. In general, the expected outcome of the method is a neural network interpreter capable of explaining the reasoning behind another neural network's decision-making in natural language. The procedure consists of four steps:

1. In the first phase, I train a convolutional neural network (CNN), referred to as *target neural network*, using the MNIST training dataset to predict the labels of a given image input. Applying traditional methods for interpretability, I extract human interpretable features, giving insight into the functionality of the target's first convolutional layer. Given these features I create two datasets: First, I generate synthetic images that are similar to those in the MNIST dataset. However, these images are specifically designed to contain the features extracted from the CNN. Each of these images is then automatically annotated with descriptions in human languages. Alongside the synthetic images, I manually annotate a small subset of the MNIST test set. The key here is that the annotations focus solely on the specific features that the first convolutional layer is identifying. This manual process ensures that

real-world data (MNIST test images) is also represented in the interpretability study. The final step is to merge these two datasets into a single training set for the neural interpreter.

2. In the second phase, a pre-trained large language transformer model forms the basis for the *neural network interpreter* since this model already has the capacity for understanding and generating human language. The interpreter undergoes further unsupervised fine-tuning: The synthetic and the manually annotated image input are processed by the target's first convolutional layer. Afterward, the activation maps computed by the target's first convolutional layer are flattened and concatenated. Furthermore, the concatenated activation vector is character-encoded to reduce its size, making it suitable for input into a large language model.

3. In the third phase the interpreter is trained to predict the human interpretable features (extracted from the first phase) given the target's first layer activations for a given image in an autoregressive manner. This ensures that the interpreter is not only learning to associate the activations with the descriptions but is also learning the patterns within the activations themselves. Thus, this fine-tuning phase enables the interpreter to articulate the rationale behind the target CNN's classifications.

4. In the final phase, I assess the interpreters's capability to explain the features represented in the activations of the target using a held-out set of synthetic data as well as a held-out subset of the MNIST test dataset.

Since this experiment is a proof-of-work concept conducted by a single person, I selected the MNIST dataset for its simplicity. I assume that more complex and diverse data sets such as CIFAR10, medical images, or even the intricate activations of large language models would make the interpreter much more robust and enable it to generalize. This assumption is consistent with the literature, where a certain amount of diverse input data was required as a threshold for unsupervised learning, after which the large language models began to show emergent interesting capabilities [Rad+19].

A natural question to ask would be why I would not simplify this design by skipping the target neural network and directly training the neural network interpreter to detect and communicate the feature in the MNIST image. The crucial point is that we want this method to also work for huge intricate datasets where the target neural network learns these features through an extensive and complicated training process. In consequence, the target's activations represent valuable features storing condensed knowledge often extracted from millions of data points and thus are much more informative about the data than the predicted labels or the raw input of a single image alone. Moreover, expecting a single neural network (the interpreter) to simultaneously learn to detect intricate features from raw data and effectively communicate them is optimistic. Instead, the work is split between the target and the interpreter neural network. Furthermore, this indirect approach

could also be used in a follow-up study to understand and communicate more abstract features represented in later layers.

5.2 EXPERIMENTAL SETUP AND RESULTS

In this section, we describe the experiment across four stages on a technical level. Initially, in Subsection 5.2.1, the focus is on outlining the target neural network and the process for gathering the synthetic and manually annotated images. Moving forward, Subsection 5.2.2 describes the neural interpreters and the construction of its training and validation dataset. The fine-tuning of the neural interpreters is then elucidated in Subsection 5.2.3. Lastly, we cover the evaluation of the interpreter's performance in Subsection 5.2.4.

5.2.1 *First Phase: Target Neural Network and Data Collection*

Figure 5.1 depicts the target neural network consisting of two convolutional and one linear layer: The first convolutional layer features 1 input channel for the 28×28 pixel gray-scale training images. It has 8 output channels, using a 9×9 kernel, a stride of 1, padding of 4, ReLU activation and a 2×2 MaxPool. These filters generate activation maps of size 14×14 that are collected and will be analyzed by the neural interpreter at a later stage. The second convolutional layer expands the channels from 8 to 16, maintaining the same kernel size, stride, padding, ReLU activation and MaxPool. The network concludes with a fully connected layer, transitioning to 10 output units corresponding to the MNIST data set classes. As a first step, the target is trained on the MNIST training set, reaching an accuracy of 98 percent on the test set (Experiment 10).

The large kernel size of the first convolutional layer is unusual but aims to strike a balance between interpretability and classification accuracy since larger kernel sizes are more accessible to interpretability techniques, namely weight visualization, filter visualization, and DeepDream methods. Applying these techniques in Experiment 11, I manually interpreted the convolutional layers and discovered low-level concepts in the first convolutional layer of the target neural network. Note that this result is consistent with research showing that convolutional neural networks store low-level features in early layers [MOT15]. Detected concepts are:

1. diagonal, contra-diagonal, horizontal, and vertical lines.
2. patchy textures, contrasts, corners, checkerboard patterns.
3. swirls, curves. ellipses
4. crossings and interconnected structures.

We extract these shapes because the interpreter can only learn features present in the target CNN's activations. If manual inspection reveals that the CNN computes features like diagonals and verticals, we should expect a competent interpreter to detect them. However, we can't expect it to predict features

not stored in the activations; for example, we can't demand circle prediction if the CNN lacks filters to compute them.

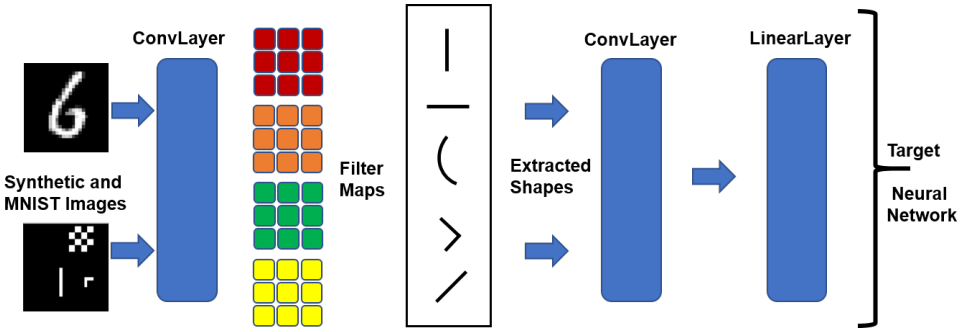

Figure 5.1: Mapping synthetic and MNIST images to activations: This diagram showcases how activations (colored boxes) within the target neural network are sourced and highlights that these activations store information about distinct primitive shapes and patterns extracted through traditional visualization techniques. Note that the second convolutional layer and the linear layer are only crucial for training and testing the *target neural network* but are not used at a later stage.

Thus, we construct the synthetic and manually annotated dataset around the detected features. For the manually annotated dataset (Experiment 12), I extracted the first 100 images of the MNIST test set and human-annotate these images with a structured description. The description includes the number of shapes present in the picture and the specific number of every shape extracted in Experiment 11 respectively.

Similarly, I generate 15,000 28x28 grey-scale MNIST-like synthetic images containing the above extracted shapes (Experiment 13). These synthetic images contain these shapes with a random orientation, length, and start and end point. This approach ensures a diverse dataset. The number of shapes in the image follows an exponential distribution to favor a lower number of shapes. In consequence, the shapes tend to clutter less, enabling better interpretability. Every image is automatically annotated with a description containing the number of all shapes and the number of every single shape, like the synthetic images. Figure 5.2 presents examples from the image collection, showcasing a synthetically generated image and an MNIST image alongside their corresponding structured descriptions for clarity. One percent of the synthetic dataset was allocated for validation purposes, while the remainder synthetic images and the manually annotated images were utilized during the training phase of the experiment.

5.2.2 *Second Phase: Neural Interpreter and Dataset Collection*

We test two large language models assuming the role of the neural interpreter. The first large language model acting as the interpreter is *Mistral7b* . This 7.3 billion parameter model surpasses the capabilities of much larger models such as Llama 2 13B [Roz+23] employing advanced attention mechanisms such as grouped-query [Ain+23] and sliding window attention [BPC20].

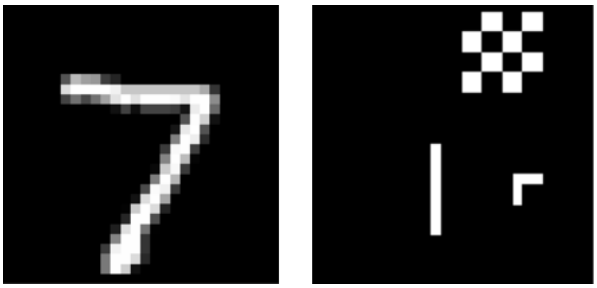

Figure 5.2: The left image, sourced from the MNIST test dataset, is labeled with the description *Image with 3 shapes: 1 corner, 1 horizontal, 1 diagonal*. In contrast, the synthetically generated image on the right carries the annotation *Image with 3 shapes: 1 corner, 1 vertical, 1 checkerboard*.

Mixtral8x7b, acting as the alternative neural interpreter, is a sparse mixture of expert models [Gal+22]. Pre-trained on web-extracted data, it employs a router network to select two out of eight experts. It has 46.7 billion total parameters but uses only 12.9 billion per token during inference. Mixtral8x7b outperforms Llama 2 70B [Roz+23] in some benchmarks, supports multiple languages, and excels in code generation. Both models descend from the decoder-only transformer architecture presented in Chapter 4. Whereas Mistral7b has a context length of 8k tokens, Mixtral8x7b can handle up to 32K tokens. Released under Apache 2.0, both models are freely usable.

The decision to utilize large language models as neural interpreters is driven by the aim of leveraging their capabilities across a variety of domains achieved by extensive pertaining. As argued in Section 5.1, we assume that these large language models, when used as neural interpreters, do not start from scratch but have an intuitive understanding of geometric shapes, pattern detection strategies, and neural networks and can understand and adhere to English instructions. This significantly simplifies the task at hand. Instead of needing to train a model from the ground up to understand and interpret complex numerical patterns, these models only require fine-tuning. The fine-tuning process is focused on teaching the models to apply their pre-existing abilities to a new task: translating numerical patterns, specifically those derived from neural network activations, into understandable human language. The analogy here is teaching someone with a background in linguistics how to interpret a new language rather than teaching someone who has never studied languages before.

Next, we build the dataset to fine-tune the pre-trained model: First, we pass each image through the first convolutional layer of the target CNN and collect the activations stored in the filter maps of the first convolutional layer as depicted in Figure 5.1. The activations are flattened and concatenated, resulting in high dimensional vectors. Specifically, since we have eight activation maps containing 14×14 entries, each image is converted into a vector of dimensionality $1,568$. Because the activations are composed of FP32 (32-bit floating point) numbers, and the language model's tokenizer is designed primarily for text rather than numbers, directly tokenizing the activation vectors leads to excessively long sequences. Thus, I explored the activation's

distribution of every filter and developed a char-encoding scheme using binning to significantly condense the data representation from approximately 20,000 to around 1,000 tokens but still not loosing too much information (Details in Experiment 13). In general, one training instance consists of a prompt explaining the task and triggering the model for the neural interpretation task, the input sequence containing the flattened, binned, and char-encoded activation vectors, and the output containing the description of the shapes in the image in natural language. To enhance interpretability, the character-encoded values from each activation map were segmented using the delimiter character 'f' for a filter. This approach was designed to aid the model in identifying the individual activation maps. The data was further converted to a JSONL format and subsequently tokenized. Post-tokenization, sequences were uniformly padded to a length of 1,280 tokens. For illustration purposes, an example of a training instance before padding is provided below. (see Example 5.2.2). This instance demonstrates the structure of the prompt, input sequence, and expected output.

Listing 5.1: Example Training Instance

```
<s> You will be presented with sequences of characters representing
    features of images, which could be basic geometric structures or
    elements similar to MNIST handwritten digits. These sequences are
    encoded representations of the image's activation maps, derived from
     the first layer of a convolutional neural network. The character 'f
    ' marks the beginning of a new filter's activation map of the first
    layer. The characters 'z', 'l', 'm', 'h', and 'v', indicate varying
    levels of activation: zero, low, medium, high, and very high,
    respectively. Your task is to understand the patterns in the
    sequence and to interpret these sequences to determine the number
    and types of shapes present in the image. The desired output format
    should be 'output: Image with [number of shapes] shapes: [number of
    first shape] [name of shape] ... [number of last shape] [name of
    last shape]', where the description includes the types of shapes,
    such as corners, edges, ellipses, crossings, checkerboard pattern,
    horizontals, verticals, etc.
### Input: fzlzzzzzzzzzzlllzzllzzzzzzlllzllzzhhlllllllllzlvvm
    lmmllllllzlvvlzmlzzlllzmhhzzlhlllllllzhmzhvhlmllll ...
    zlzzzzmmzvhzzzzlzzzzzzmvzzzzzmzzzzzzvvzzzzzmzzzzzlvhzzzzz
### Output: Image with 10 shapes: 1 corner, 1 vertical, 2 horizontals, 2
     diagonals, 1 crossing, 1 checkerboard, 2 ellipses</s>
```

Figure 5.3 visualizes the complete data pipeline for the neural interpreter's training set for autoregressive training. In summary, we pass the synthetic and MNIST test images through the first convolutional layer, collect the activations, concatenate them, apply binning to transform them into character sequences, prepend the instruction prompt, and ground truth description of the image.

In this training methodology, the process begins with all values, except the start-tokens, being masked. Initially, the model's task is to sequentially predict the task description embedded in the prompt. Once the model suc-

cessfully predicts the task description, the prompt remains fully unmasked, but the subsequent activations and the ground truth description are still masked. At this juncture, the model is challenged to predict the activations of the first filter based solely on the information provided in the prompt and the already predicted activations in the first filter. Once it accomplishes this, it proceeds to predict the activations of the second filter. However, this time, it has more information at its disposal – the prompt and all activations of the first filter. This incremental unmasking and prediction continue for each subsequent filter. This step-by-step revelation of information is not just about the model learning to predict the activations correctly; it's about understanding the patterns within these activations and, more importantly, grasping the relationships between them. The idea is that through this process, the interpreter starts to discern the abstract, semantic concepts encoded in the activation maps by the target neural network.

Finally, after the model has predicted the last activation of the last filter, the prompt and all the activations from all the filters are now unmasked. However, the ground truth description of the shapes in the images remains masked. The task for the model now is to bridge the gap between the numerical and linguistic representations; it must predict the ground truth descriptions, translating the features stored in the numerical activation patterns into the descriptions that store the same features in human language.

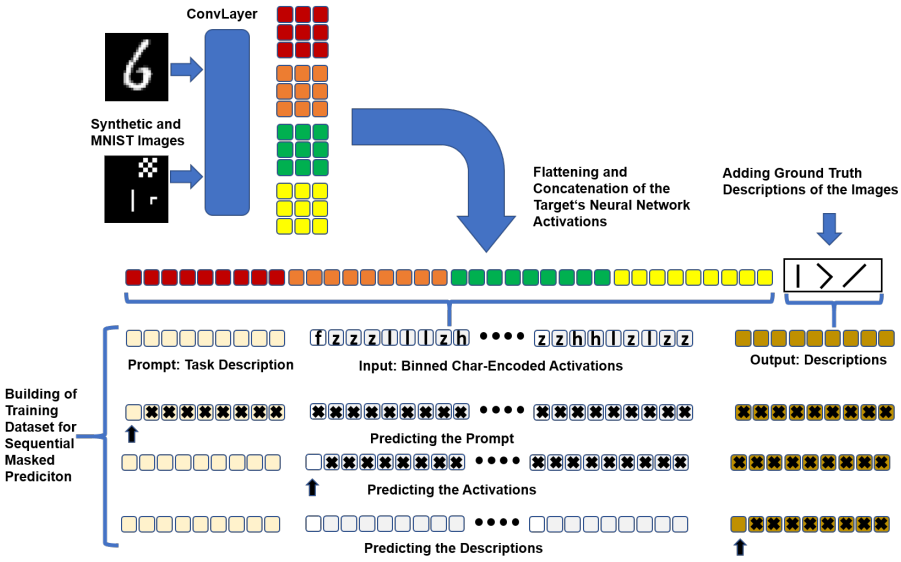

Figure 5.3: Overview of the Data Pipeline for the Neural Interpreter's Training Set

5.2.3 *Third Phase: Fine-Tuning the Neural Interpreter*

For fine-tuning, I set up a configuration using 9 A40 GPUs with 48 GB of GPU memory each, spanning over a period of one to two days for the first model and three days for the second model. To enable parameter efficient fine-tuning of a large language model, I incorporated techniques like *double*

4-Bit quantization [Det+22] and *Low-Rank Adapters* [Hu+21] to decrease the memory footprint during training. Whilst decreasing the costs for training these techniques do not degrade performance [Det+22] [Hu+21]. Particularly, I demonstrate in Experiment 13 that Low-Rank Adapters reduce the number of trainable parameters to around two percent as compared to the original model size, dramatically decreasing the number of the optimizer state memory footprint during training.

I used the *brain floating point 16* (bloat16) datatype [Kal+19] for fine-tuning Mistral7b. Kalamkar et al. show deep learning training using bfloat16 tensors often achieves the same state-of-the-art results across domains as FP32 tensors in the same number of iterations and with no changes to hyperparameters [Kal+19]. Additionally, I used paged optimizers to prevent memory spikes during gradient checkpointing from causing out-of-memory errors. In this vein, paged optimizers move the memory of optimizer states to the CPU when the GPU runs out of memory [Det+23]. Hyperparameters for the fine-tuning procedure were set according to current recommendations [Det+23].

Prior to full-scale training, extensive debugging and monitoring were integral parts of the training preparation. This included verifying the distribution of the activations, the construction of the training instances, the insertion of LorA adapters next to all matrices in the attention mechanism, ensuring uniform sequence lengths post-padding, and monitoring GPU utilization. The use of the Huggingface Trainer facilitated the management of multi-GPU training. The training of Mistral7b was limited to two and a half epochs and the training of Mixtral8x7b to two epochs. This decision was also influenced by signs of overfitting starting to appear in the model's performance. Additionally, training the model for more than three periods led to severe parameter degradation and loss explosion that the model did not recover from.

5.2.4 *Results*

To comprehensively interpret the results, it is essential to first acknowledge the complexities encountered by the neural interpreter. The methodology involves flattening and concatenating the activation maps, a process which inherently strips away spatial information pertaining to individual activations. Further complexity is added by the binning of activations, contributing additional noise to the dataset. The resolution of the images under consideration, limited to 28x28 pixels, presents another layer of difficulty. For instance, such a low resolution hampers the ability to accurately distinguish between round shapes and angular corners, leading to potential ambiguities in the interpretation process. Moreover, it is only assumed that the shapes the interpreter must predict are stored in the activations, as this inference is based on the currently available approximate visualization techniques and human intuition. Another notable aspect of the dataset is the introduction of human-described images. The annotations, being subject to individual interpretation, can vary significantly between different annotators and even

exhibit inconsistency when the same annotator repeats the task. This subjectivity adds a layer of unpredictability to the data. First, we analyze the loss and validation metrics as displayed in Figure 5.4. Both models' training and validation losses significantly decrease over the course of two epochs, indicating effective learning and generalization. Mistral7b's validation loss decreases more smoothly, suggesting a more stable fine-tuning process compared to the variability observed in Mixtral8x7b's performance. In sum, convergence of all loss metrics suggests successful optimization of both models.

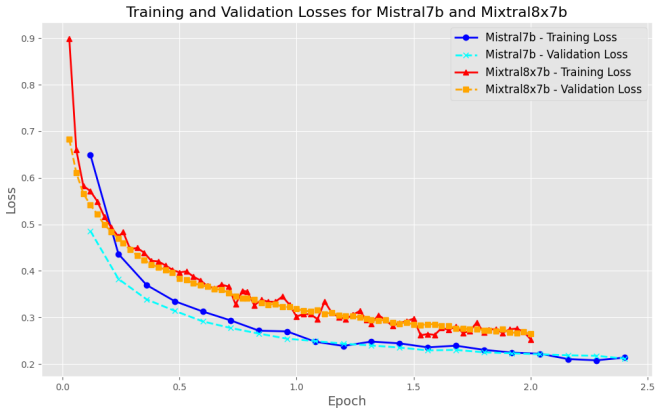

Figure 5.4: Loss curves of Mistral7b and Mixtral8x7b models during fine-tuning, showcasing the epoch-wise progression of training and validation losses.

5.2.4.1 *Performance of Mistral7b*

Experiment 14 implements the evaluation of Mistral7b where we investigate the interpreter's response to the activations of synthetic and MNIST images. For this purpose, we generate 20 new synthetic images. These images are further processed to gain the char-encoded input activations. The input is prepended with the task description to trigger the trained model behavior. The fine-tuned model is loaded and successively prompted with these messages. The model's output is compared to the ground truth, that is the number of shapes present in the image and the number of every types shape. Out of 20 images, the model accurately described the correct number of shapes 16 times, and in 13 of these instances, it also correctly predicted the number of each individual shape. Moreover, it missed three times only a single shape and two times it predicted at least one correct shape. Notably, the model was always correct in its prediction when only a singular shape was present. However, I observed that the model consistently continued to hallucinate further shapes even after a correct description. Additionally, the model regularly continued to hallucinate char-encoded activations after the image description.

Next, I evaluate the interpreter on MNIST data. 100 images from the MNIST test set, specifically indexed from 100 to 200, were utilized. This ensures that both the primary target CNN and the interpreter model were

tested on images they had not encountered during their training phase. Given that the interpreter encountered only a small fraction (less than one percent) of the MNIST dataset during the training phase, suggests an application of learned skills to a somewhat novel task allowing us to assess the model's generalization capabilities beyond its primary training context. In Experiment 14, I display the original MNIST image alongside the interpreter's description. The model shows a greater tendency to recognize multiple shapes in MNIST images compared to synthetic ones. While synthetic images seldom feature more than four shapes, the interpreter identified up to 14 shapes in some of the MNIST numbers. This suggests that the interpreter recognizes the numbers as having more complex geometric structures than those found in synthetic images. Interestingly, the model was precise in recognizing distinct shapes including the crossings occurring in numbers such as two or eight. Moreover, the model can generate detailed descriptions for some images, demonstrating its capacity for a more comprehensive analysis. However, limitations are more pronounced compared to synthetic images. It frequently misses simple shapes or produces character-encoded activations. Without specific task directives and the delimiter ### *Output: Number of Shapes:*, the model always outputs character-encoded activations. The inclusion of this delimiter prompts the model to switch to generating descriptions however only after repeating the same delimiter again clearly violating the instruction format. In further experiments the fine-tuned model ignored instructions to reply with a confidence score and was not able to explain its decisions. Interestingly, when I performed the same experiment with ChatGPT-4 (see Experiment 15), that is prompting the model with the task description and the char-encoded activation vectors, it was able to understand the task and predict (albeit wrong) descriptions. The crucial point however is that ChatGPT-4 was able to generate sensible explanations for its decisions. This suggests that the same experiment, done with a stronger model would yield better results. Hence, I now investigate the performance of Mistral's successor model Mixtral8x7b.

5.2.4.2 *Performance of Mixtral8x7b*

In Experiment 15, I perform the evaluation of the Mixtral8x7b model in its pre-fine-tuned state. Subsequently, in Experiment 16, I evaluated the same model after fine-tuning, using a similar evaluation methodology as employed in the assessment of Mistral7b. For the 20 synthetic images, the model, prior to fine-tuning, did not generate a description. It merely perpetuated the existing pattern of activations. In particular, the model ignored the instructions from the prompt and the specified answer format, and simply extended the activation sequence with the character 'z', which was the predominant character-encoded activation in the input sequence.

In contrast, Mixtral8x7b post-fine-tuning successfully predicted the correct number of shapes 16 times. On 11 occasions, it accurately identified not only the total number of shapes but also the exact count of each individual shape. Additionally, the model missed only a single shape in eight instances.

I also noticed a reduction in hallucinations compared to the previous model. This improvement manifested as the model ceasing to generate shapes and instead continuing the next sequence in the desired format ### *Input: f...* or sometimes outputting the activation character 'z', which corresponds to a zero. Interestingly, the model occasionally followed the instruction to assign a confidence score to its prediction. In some cases, after making a prediction, the model began discussing the context of the sequence, stating that it is a fictitious dataset created for educational purposes. Like the previous model, Mixtral8x7b was always correct in its prediction when only a singular shape was present, misattributions only occurred when multiple shapes were present.

Additionally, I evaluated the trained model on the same 100 images from the MNIST test set used for assessing Mistral7b. Before fine-tuning, the model generated generic descriptions unrelated to the actual images. For example, it described various MNIST images depicting the numbers eight, three, seven, four, and nine with a generic output like 10 squares. Post-fine-tuning, however, the model demonstrated improved performance. It began to guess the number of shapes in each image and provided a description that included both the number of shapes and the types of shapes present. Notably, while Mistral7b often exhibited erratic behavior, such as outputting activations instead of descriptions, the post-fine-tuned Mixtral8x7b model consistently recognized that it should output descriptions. These descriptions, predominantly consisting of shapes, were not very precise and tended to be similar across different numbers, indicating that the model struggled when dealing with a greater number of shapes than it was trained on. On a side note, the model also started to add comments. For instance, in one case after outputting the image descriptions, the model independently decided to add a note, stating that the input sequence (meaning the activation pattern) may contain additional information that can be ignored for this problem and specifies that there are some relevant parts to identify.

In tests to assess the model's ability to not only predict shapes but also discuss them naturally, several interesting observations were made. For instance, the model was asked about the presence and number of curve. Since this shape was not a part of the training set, this question inquired about the model's ability to generalize. In this context, before fine-tuning, the model tended to give brief responses, often stating that there were no curves in the image. This changed significantly after fine-tuning. The model began to use the shapes it had learned during fine-tuning to reason about the presence of curves. For example, it suggested that certain sequences indicated a horizontal ellipse shape and that an increased number of points might signify diagonal lines. It also claimed to observe checkerboard patterns, attributing these to a detectable repetitive nature along axes, thus imitating an explanation using the learned features. Importantly, the model only exhibited the behavior if the prompt included a specific hint to include the knowledge it has about the preceding input sequence that it learned during the fine-tuning procedure. Without this hint, the interpreter's performance

degraded severely. However, these post-fine-tuning answers were most of the times inconsistent, generic and lacked clarity.

A notable change was observed in how the model responded to a short char-encoded sequences of activations. Before fine-tuning, given a sequence like ### *Input: fzllzz*, the model would output a nonsensical reply such as a numerical value. After fine-tuning, the model continued the sequence by producing activation patterns indistinguishable from the activation pattern from regular images. This indicates that the model had learned the inherent structure of activation patterns. Furthermore, when the model was inquired about the origin of a provided sequence activation pattern the model after fine-tuning could explain that it originated from a convolutional layer, which the pre-fine-tuned model did not achieve. This suggests that the fine-tuning process enhanced the model's understanding and its ability to provide more sophisticated explanations about its internal processes of the target neural network even though strong limitations were observed.

5.2.4.3 *Overall Conclusion*

The primary aim of this experiment was to demonstrate the feasibility of converting information contained in a sequence of activations into human-readable language. The results demonstrating the interpreter's capacity to reliably translate an activation pattern into the correct description in terms of shapes clearly indicate that this objective has been met. However, given that this was a preliminary study, the achievement was realized within a highly constrained context. In this context, I noted that the performance of the interpreters was extremely sensitive to the specifics of the prompts given. The presence of an extra space, for instance, could drastically shift the response from being nonsensical to providing meaningful insight. Therefore, the remainder of this chapter will focus on discussing the insights gained from the experiment, highlighting further limitations, and exploring potential research directions and improvements for future studies.

5.3 LIMITATIONS AND FUTURE WORK

The idea of automated neural network interpretability and the successful implementation of a neural network interpreter illustrates the possibility of using artificial intelligence to decipher complex data patterns and translate them into human-interpretable formats. However, the study also uncovers inherent limitations.

5.3.1 *Representation of Activations*

We can conceptualize our setting as an *activation-to-language* translation task, similar to other modality translation tasks like *text-to-text*, *image-to-language*, *speech-to-text*, etc. By flattening, concatenating, binning and char-encoding the activations, the rich and complex information originally present in the

data is compressed into a linear, text-based format to make it suitable for the analysis by the interpreter. In this context, applying a multi-modal large language model to interpret neural activations is a compelling idea because it aligns better with the model's inherent capability to handle different data representations. When we flatten and combine these activations, the interpreter loses the ability to process the positional context that was originally encoded in the filter maps. The capabilities of future models with larger context sizes present an opportunity by allowing us to append positional vectors, which helps in preserving the spatial context. As we move towards analyzing activations from even larger neural networks, the high-dimensional nature of these activation vectors becomes a challenge. In this vein, we consider the use of auto-encoders to extract smaller feature vectors which the interpreter is able to process further.

5.3.2 *Making a Better Dataset*

The results of this study show that the interpreter was very reliable on the synthetic dataset but this reliability decreased substantially on the MNIST dataset. The observation that both Mistral7b and Mixtral8x7b performed well when identifying few synthetic shapes in contrast to identifying multiple shapes arranged in complex ways to construct numbers points to a number of limitations in the training dataset. For example, a corner might not be only interpreted as a corner but also differently based on its orientation - as an intersection of a horizontal and a vertical line, or as a diagonal meeting a contra diagonal. Additionally, the low resolution of the 28×28 images makes it difficult to discern, for instance, corners from curves.

Another problem arises from the subjective nature of human annotations for the MNIST dataset. These annotations, based on personal intuition, can lead to inconsistencies in the data. On a related note, the structured descriptions created as ground truth were basic and lacked complexity. These automatically generated descriptions were in a structured format, and during the fine-tuning process, the model was specifically trained to produce outputs that adhered to this structured format. As a result, the model became highly specialized in outputting these structured descriptions and less flexible in generating more natural, free-flowing language. This rigidity posed a challenge when attempting to guide the model into having more natural and conversational discussions about the task and the input sequence of neural activations later on. Essentially, because the model was so finely tuned to produce structured responses, it struggled to adapt to a different style of output that required a more open-ended, narrative form of expression. Additionally, to address the subjective bias in the MNIST dataset, employing a diverse set of annotators would ensure a more balanced and comprehensive range of human interpretations.

These limitations suggest that the training dataset was not complex and diverse enough. There are a number of possibilities to increase the quality of the dataset. Maybe counterintuitively, using larger images with more details

would make it easier for the interpreter to detect discernible patterns in the resulting activations. Once trained on this more diverse dataset, the model is likely to demonstrate improved generalization capabilities. Moreover, we could use large language models themselves to expand the generic, structured, and automatically generated descriptions into more natural-sounding, elaborate, and nuanced descriptions in an automated way.

5.3.3 *Refining the Target, the Interpreter, and the Training Procedure*

The experiment indicated that synthetic data helped the neural interpreter to understand the relationship between neural activations and the shapes represented within them. In fact, the large quantity of synthetic data seems to be necessary to steer the interpreter, originally trained to map text to text, to associate input sequences predominantly comprising numerical values with the descriptions. Thus, the continued use of synthetic data and the adoption of a pretraining approach to familiarize the neural interpreter to understanding activations an attractive avenue for further refining the methodology. The key idea is to view the target neural network's activations as statistical distributions that the interpreter aims to learn. By generating synthetic data that mimics the format of encoded activations from a variety of stochastic sources, such as Gaussian or uniform distributions, we can generate a generic comprehensive dataset that prepares the large language model to act as an interpreter. Specifically, we conduct a second pretraining where the interpreter learns to recognize and describe the characteristics of these synthetic activation patterns. For instance, it might learn to identify and articulate whether a given activation pattern resembles a Gaussian distribution with specific parameters or if it aligns more closely with a uniform distribution.

The intent behind this approach is to equip the interpreter with a foundational understanding of data properties and statistical variations to enhance the interpreter's ability to discern and interpret complex patterns in actual neural network activations. When subsequently exposed to real activation data, the interpreter can leverage this foundational knowledge to provide more insightful and accurate interpretations. This approach aligns with established machine learning principles, particularly the emphasis on the importance of diverse and extensive training datasets during the pretraining phase in enhancing a model's ability to generalize and adapt to new data [RJS17].

The interpreter's ability to transition from outputting activations to generating meaningful descriptions, instead of merely continuing the char-encoded activation patterns without inherent understanding of the sequence, showcases its strong capabilities. Notably, this shift, particularly after receiving an input prompt followed by hundreds of char-encoded activations, highlights the interpreter's effective functionality of understanding the full context of its input. However, it was also noted that the interpreter frequently extended activations or shapes even after accurate predictions. This tendency was no-

ticeably reduced in the larger model, suggesting that more advanced foundational models are less prone to such *hallucinations*. However, this behavior could also stem from the autoregressive nature of the fine-tuning process, which inclines the model to behave like a document completer, continuing to generate data that mirrors the learned sequences.

Another area of improvement could be directed towards altering the target neural network making it more accessible to the interpreter. For instance, training the network with sparser hidden layers or setting these layers to a lower floating point precision might simplify the interpreter's task. Such adjustments could potentially eliminate the need for binning techniques currently used to manage the size of the activation vector. Additionally, this study focused on the first convolutional layer of the target network, chosen based on the low-level features observed through visualization techniques. However, to capture more complex descriptions or patterns, it might be beneficial to vary which network layer is analyzed and empirically determine which layer yields the best performance for the interpreter.

5.3.4 *Translating the MNIST Setting into Practical Use Cases*

The original idea for proposing the neural interpreter was to employ it as a tool for explaining the decisions made by machine learning models in biological settings. For instance, even though convolutional neural networks analyzing images captured during endoscopic procedures, are effective in cancer detection [Zha+22] it is unclear how these models reach their conclusions. In this context, the utility of a trained neural interpreter becomes evident. It would analyze the activation patterns within the convolutional neural network, which in this scenario serves as the target model. The interpreter's role is to translate these activation patterns into understandable human language. Potentially, the neural interpreter could examine these activations and provide descriptions that highlight the presence of highly irregular geometric patterns. For instance, it might identify an unusually high number of distinct shapes, implying a high presence of cells appearing markedly different from their neighbors in images that are identified as cancerous. When analyzing non-cancerous images, the interpreter would produce a lower number of unique shapes and less elaborate descriptions. This information would reveal that the target CNN is likely focusing on regions within the images where cellular abnormalities are present to determine whether the image signifies cancer. Essentially, the interpreter helps to uncover that the CNN is using these areas of irregularity as critical indicators for classifying an image as cancerous.

This methodology is related to the idea of using the interpreter as a tool for hypothesis testing in biological research. We rephrase the previous concept to illustrate the idea with a simple example: Practitioners in biological fields formulate hypotheses about certain patterns or features that might be present in the data. For instance, researchers might hypothesize that the increased frequency of mitotic figures in tissue samples, a consequence of

more frequent cell division in cancer cells compared to healthy cells, serves as a reliable predictor for breast cancer [Bie+94].

To test this hypothesis, a target neural network is trained on endoscopic images to predict cancer. Separately, a neural interpreter is developed. Its job is to analyze how the neural network makes its decisions by examining the network's internal activation patterns. The interpreter is specifically trained to detect and describe the hypothesized patterns – for example, it might be trained to recognize and report the number of cells that are performing mitosis. The crucial step comes when the interpreter is applied to new, unseen endoscopic images. If an increased number of mitotic figures is indeed a significant indicator for cancer, as hypothesized, the neural network will likely have incorporated this pattern as a key feature. This pattern will be reflected in the network's activation patterns. The neural interpreter, trained to identify these specific patterns, should consistently report high mitotic counts in the endoscopic images corresponding to sick individuals and a lower mitotic count in healthy ones. Conversely, if the hypothesis is not valid, the neural network will not use these patterns (repeated base pairs) for its diagnosis. Consequently, these patterns won't be present or prominent in the network's activation patterns. The interpreter, relying on these activation patterns to make its assessments, will not be able to consistently identify the hypothesized pattern. It will show inconsistency in its reporting, reflecting the absence of these patterns in the activations of the neural network when analyzing the endescopic images showing cancerous tissue.

Note that the here proposed approach of automated neural interpretability is extremely generic making it applicable in various domains including the research around superalignment, as mentioned in the introduction. In this vein, a possible application involves using the automated neural interpreters to understand the internal workings of large language models themselves with a two-step process: First, a target large language model generates a detailed text on a specific subject, (for instance a detailed architectural description of Notre-Dame or a poetry about a magical ephemeral night scene). Alongside, an automatic summary of this text is generated, serving as a condensed representation of the main ideas. In a second step, the activations of some chosen layer of the target large language model during the text generation are collected. A separate neural interpreter is then employed to predict the summary based on these activations. Essentially, this interpreter reads the internal state of the target large language model and decodes what the model was thinking or focusing on during its generation task. A neural interpreter of this kind can be employed to directly monitor and oversee the internal workings of superhuman AI models. Its role would be to ensure these models stay true to human objectives and do not develop power-seeking tendencies [Car22].

A logical question that arises is the trustworthiness of using one machine to monitor another. To address this, the activations of the target AI model can be collected, duplicated, and distributed to various interpreters that are isolated from communicating with the target or each other. Consistency in

their interpretations would indicate the interpreters' reliability. Moreover, to further test the interpreters' accuracy, the activations sent to them could be subtly altered to include covert messages that appear to come from the target neural network. Observing whether the interpreters detect and report these hidden messages can provide additional insight. Even if the target neural network is capable of embedding sophisticated hidden messages in its activations, the interpreters cannot definitively determine if these originate from humans or the AI itself. As a result, some interpreters may mistakenly interpret these messages as human input and report them with greater frequency.

Looking to the future, the upcoming years are likely to witness the development of increasingly sophisticated open-source foundational models. As shown in this experiment, these large models can be fine-tuned with a relatively modest computational budget, making them accessible for use by a diverse range of researchers. This leads to the question of how large language models, which are increasingly resembling artificial general intelligence, can be applied in fields that directly benefit humanity, such as biological sciences. The experiment described here suggests a research direction where these advanced AI models can become valuable tools.

BIBLIOGRAPHY

[Agg18] Charu Aggarwal. *Machine learning for text: An introduction*. Springer, 2018.

[Agg20] Charu Aggarwal. *Linear Algebra and Optimization for Machine Learning*. Springer, 2020.

[Agg23] Charu Aggarwal. *Neural Networks and Deep Learning: A Textbook*. 2nd ed. Springer, 2023.

[Ain+23] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. "GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints." In: *arXiv preprint arXiv:2305.13245* (2023).

[Ala18] Jay Alammar. *The Illustrated Transformer*. Blog post. Accessed: 2023-11-28. 2018. URL: http://jalammar.github.io/illustrated-transformer/.

[Amo+16] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, et al. "Deep speech 2: End-to-end speech recognition in english and mandarin." In: *International conference on machine learning*. PMLR. 2016, pp. 173–182.

[Api+21] Andrea Apicella, Francesco Donnarumma, Francesco Isgrò, and Roberto Prevete. "A survey on modern trainable activation functions." In: *Neural Networks* 138 (2021), pp. 14–32.

[BKH16] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. *Layer Normalization*. 2016. arXiv: 1607.06450 [stat.ML].

[BCB16] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. 2016. arXiv: 1409.0473 [cs.CL].

[Bah+21] Yasaman Bahri, Ethan Dyer, Jared Kaplan, Jaehoon Lee, and Utkarsh Sharma. *Explaining Neural Scaling Laws*. 2021. arXiv: 2102.06701 [cs.LG].

[Bel15] Richard Bellman. *Adaptive Control Processes-A Guided Tour (Reprint from 1961)*. 2015.

[BPC20] Iz Beltagy, Matthew E. Peters, and Arman Cohan. *Longformer: The Long-Document Transformer*. 2020. arXiv: 2004.05150 [cs.CL].

[Ben12] Yoshua Bengio. "Practical recommendations for gradient-based training of deep architectures." In: *Neural Networks: Tricks of the Trade: Second Edition*. Springer, 2012, pp. 437–478.

[BB99] Yoshua Bengio and Samy Bengio. "Modeling high-dimensional discrete data with multi-layer neural networks." In: *Advances in Neural Information Processing Systems* 12 (1999).

[Ben+03] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. "A Neural Probabilistic Language Model." In: *Journal of Machine Learning Research* 3 (2003), pp. 1137–1155.

[BSF94] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. "Learning long-term dependencies with gradient descent is difficult." In: *IEEE transactions on neural networks* 5.2 (1994), pp. 157–166.

[BB12] James Bergstra and Yoshua Bengio. "Random search for hyper-parameter optimization." In: *Journal of machine learning research* 13.2 (2012).

[BK23] Saumyamani Bhardwaz and Jitender Kumar. "An Extensive Comparative Analysis of Chatbot Technologies-ChatGPT, Google BARD and Microsoft Bing." In: *2023 2nd International Conference on Applied Artificial Intelligence and Computing (ICAAIC)*. IEEE. 2023, pp. 673–679.

[Bie+94] Stefan Biesterfeld, Ingrid Noll, Egbert Noll, Dieter Wohltmann, and Alfred Böcking. "Mitotic Frequency as a Prognostic Factor in Breast Cancer." In: *Human Pathology* 26.1 (1994), pp. 47–52. DOI: 10.1016/0046-8177(95)90113-2.

[Bis95] Christopher M Bishop. *Neural networks for pattern recognition*. Oxford university press, 1995.

[BO18] Léonard Blier and Yann Ollivier. "The description length of deep learning models." In: *Advances in Neural Information Processing Systems* 31 (2018).

[Bro+20] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. arXiv: 2005.14165 [cs.CL].

[Bub+23] Sébastien Bubeck et al. *Sparks of Artificial General Intelligence: Early experiments with GPT-4*. 2023. arXiv: 2303.12712 [cs.CL].

[Bur+23] Collin Burns, Pavel Izmailov, Jan Hendrik Kirchner, Bowen Baker, Leo Gao, Leopold Aschenbrenner, Yining Chen, Adrien Ecoffet, Manas Joglekar, Jan Leike, et al. "Weak-to-Strong Generalization: Eliciting Strong Capabilities With Weak Supervision." In: *arXiv preprint arXiv:2312.09390* (2023).

[Car22] Joseph Carlsmith. *Is Power-Seeking AI an Existential Risk?* 2022. arXiv: 2206.13353 [cs.CY].

[Che+20] Mark Chen, Alec Radford, Rewon Child, Jeffrey Wu, Heewoo Jun, David Luan, and Ilya Sutskever. "Generative pretraining from pixels." In: *International conference on machine learning*. PMLR. 2020, pp. 1691–1703.

[CP12] Serena H Chen and Carmel A Pollino. "Good practice in Bayesian network modelling." In: *Environmental Modelling & Software* 37 (2012), pp. 134–145.

[CL23] Sanghyuk Roy Choi and Minhyeok Lee. "Transformer Architecture and Attention Mechanisms in Genome Data Analysis: A Comprehensive Review." In: *Biology* 12.7 (2023), p. 1033.

[CLJ20] Jean-Baptiste Cordonnier, Andreas Loukas, and Martin Jaggi. "Multi-head attention: Collaborate instead of concatenate." In: *arXiv preprint arXiv:2006.16362* (2020).

[Dat20] Leonid Datta. "A survey on activation functions and their relation with xavier and he normal initialization." In: *arXiv preprint arXiv:2004.06632* (2020).

[Daw84] A Philip Dawid. "Present position and potential developments: Some personal views statistical theory the prequential approach." In: *Journal of the Royal Statistical Society: Series A (General)* 147.2 (1984), pp. 278–290.

[Det+22] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. *LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale*. 2022. arXiv: 2208.07339 [cs.LG].

[Det+23] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. *QLoRA: Efficient Finetuning of Quantized LLMs*. 2023. arXiv: 2305.14314 [cs.LG].

[Dev+19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL].

[Elh+22] Nelson Elhage et al. "Softmax Linear Units." In: *Transformer Circuits Thread* (2022). https://transformer-circuits.pub/2022/solu/index.html.

[Fit44] Frederic B Fitch. "Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. Bulletin of mathematical biophysics, vol. 5 (1943), pp. 115–133." In: *The Journal of Symbolic Logic* 9.2 (1944), pp. 49–50.

[Gal+22] Trevor Gale, Deepak Narayanan, Cliff Young, and Matei Zaharia. *MegaBlocks: Efficient Sparse Training with Mixture-of-Experts*. 2022. arXiv: 2211.15841 [cs.LG].

[GB10] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2010, pp. 249–256.

[GBC18] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. 1st ed. The MIT Press, 2018.

[Gra14] Alex Graves. *Generating Sequences With Recurrent Neural Networks*. 2014. arXiv: 1308.0850 [cs.NE].

[Grü05] Peter Grünwald. "Minimum description length tutorial." In: *Advances in minimum description length: Theory and applications* 5 (2005), pp. 1–80.

[GR19] Peter Grünwald and Teemu Roos. "Minimum description length revisited." In: *International journal of mathematics for industry* 11.01 (2019), p. 1930001.

[HM95] Jun Han and Claudio Moraga. "The influence of the sigmoid function parameters on the speed of backpropagation learning." In: *International workshop on artificial neural networks*. Springer. 1995, pp. 195–201.

[HH23] Bobby He and Thomas Hofmann. *Simplifying Transformer Blocks*. 2023. arXiv: 2311.01906 [cs.LG].

[He+15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[Hen+20] Tom Henighan, Jared Kaplan, Mor Katz, Mark Chen, Christopher Hesse, Jacob Jackson, Heewoo Jun, Tom B Brown, Prafulla Dhariwal, Scott Gray, et al. "Scaling laws for autoregressive generative modeling." In: *arXiv preprint arXiv:2010.14701* (2020).

[Hen22] James Hennessy. *Even if they're not actually intelligent, AIs may shift the nature of human expression itself*. 2022. URL: https://www.theguardian.com/commentisfree/2022/jun/27/even-if-theyre-not-actually-intelligent-ais-may-shift-the-nature-of-human-expression-itself (visited on 11/02/2023).

[HVC93] Geoffrey E Hinton and Drew Van Camp. "Keeping the neural networks simple by minimizing the description length of the weights." In: *Proceedings of the sixth annual conference on Computational learning theory*. 1993, pp. 5–13.

[Hir+18] Toshiaki Hirasawa, Kazuharu Aoyama, Tetsuya Tanimoto, Soichiro Ishihara, Satoki Shichijo, Tsuyoshi Ozawa, Tatsuya Ohnishi, Mitsuhiro Fujishiro, Keigo Matsuo, Junko Fujisaki, et al. "Application of artificial intelligence using a convolutional neural network for detecting gastric cancer in endoscopic images." In: *Gastric Cancer* 21 (2018), pp. 653–660.

[Hoc91] Sepp Hochreiter. "Untersuchungen zu dynamischen neuronalen Netzen." In: *Diploma, Technische Universität München* 91.1 (1991), p. 31.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory." In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[Hu+21] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. "Lora: Low-rank adaptation of large language models." In: *arXiv preprint arXiv:2106.09685* (2021).

[Hua+17] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. "Densely connected convolutional networks." In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 4700–4708.

[Iof17] Sergey Ioffe. *Batch Renormalization: Towards Reducing Minibatch Dependence in Batch-Normalized Models*. 2017. arXiv: 1702.03275 [cs.LG].

[IS15] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. arXiv: 1502.03167 [cs.LG].

[Jel+77] Fred Jelinek, Robert L Mercer, Lalit R Bahl, and James K Baker. "Perplexity—a measure of the difficulty of speech recognition tasks." In: *The Journal of the Acoustical Society of America* 62.S1 (1977), S63–S63.

[JL17] Robin Jia and Percy Liang. "Adversarial examples for evaluating reading comprehension systems." In: *arXiv preprint arXiv:1707.07328* (2017).

[Jum+21] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. "Highly accurate protein structure prediction with AlphaFold." In: *Nature* 596.7873 (2021), pp. 583–589.

[Kal+19] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, et al. "A study of BFLOAT16 for deep learning training." In: *arXiv preprint arXiv:1905.12322* (2019).

[Kes+17] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. *On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima*. 2017. arXiv: 1609.04836 [cs.LG].

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks." In: *Advances in neural information processing systems* 25 (2012).

[LeC+02] Yann LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. "Efficient backprop." In: *Neural networks: Tricks of the trade*. Springer, 2002, pp. 9–50.

[LKS90] Yann LeCun, Ido Kanter, and Sara Solla. "Second order properties of error surfaces: Learning time and generalization." In: *Advances in neural information processing systems* 3 (1990).

[Lin70] Seppo Linnainmaa. "The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors." PhD thesis. Master's Thesis (in Finnish), Univ. Helsinki, 1970.

[MHN+13] Andrew L Maas, Awni Y Hannun, Andrew Y Ng, et al. "Rectifier nonlinearities improve neural network acoustic models." In: *Proc. icml*. Vol. 30. 1. Atlanta, GA. 2013, p. 3.

[MV15] Aravindh Mahendran and Andrea Vedaldi. "Understanding deep image representations by inverting them." In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 5188–5196.

[MPL15] Julian McAuley, Rahul Pandey, and Jure Leskovec. "Inferring networks of substitutable and complementary products." In: *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*. 2015, pp. 785–794.

[McM56] Brockway McMillan. "Two inequalities implied by unique decipherability." In: *IRE Transactions on Information Theory* 2.4 (1956), pp. 115–116.

[MKM19] Andreas Messalas, Yiannis Kanellopoulos, and Christos Makris. "Model-agnostic interpretability with shapley values." In: *2019 10th International Conference on Information, Intelligence, Systems and Applications (IISA)*. IEEE. 2019, pp. 1–7.

[MD91] Risto Miikkulainen and Michael G Dyer. "Natural language processing with modular PDP networks and distributed lexicon." In: *Cognitive Science* 15.3 (1991), pp. 343–399.

[MP69] Marvin Minsky and Seymour Papert. "An introduction to computational geometry." In: *Cambridge tiass., HIT* 479.480 (1969), p. 104.

[MP72] Marvin Minsky and Seymour Papert. *Perceptrons: An Introduction to Computational Geometry*. 2nd, with corrections. First edition published in 1969. Cambridge, MA: The MIT Press, 1972. ISBN: 0-262-63022-2.

[MTP15] Alessandro Montalto, Giovanni Tessitore, and Roberto Prevete. "A linear approach for sparse coding by a two-layer neural network." In: *Neurocomputing* 149 (2015), pp. 1315–1323.

[Mon+19] Grégoire Montavon, Alexander Binder, Sebastian Lapuschkin, Wojciech Samek, and Klaus-Robert Müller. "Layer-wise relevance propagation: an overview." In: *Explainable AI: interpreting, explaining and visualizing deep learning* (2019), pp. 193–209.

[Mon+14] Guido Montúfar, Razvan Pascanu, Kyunghyun Cho, and Yoshua Bengio. *On the Number of Linear Regions of Deep Neural Networks*. 2014. arXiv: `1402.1869 [stat.ML]`.

[MY16] Shay Moran and Amir Yehudayoff. "Sample compression schemes for VC classes." In: *Journal of the ACM (JACM)* 63.3 (2016), pp. 1–10.

[MOT15] Alexander Mordvintsev, Christopher Olah, and Mike Tyka. "Deepdream-a code example for visualizing neural networks." In: *Google Research* 2.5 (2015).

[NW96] Thomas R Niesler and Philip C Woodland. "A variable-length category-based n-gram language model." In: *1996 IEEE International Conference on Acoustics, Speech, and Signal Processing Conference Proceedings*. Vol. 1. IEEE. 1996, pp. 164–167.

[Nob06] William S Noble. "What is a support vector machine?" In: *Nature biotechnology* 24.12 (2006), pp. 1565–1567.

[OMS17] Chris Olah, Alexander Mordvintsev, and Ludwig Schubert. "Feature Visualization." In: *Distill* (2017). https://distill.pub/2017/feature-visualization. DOI: 10.23915/distill.00007.

[PMB13] R Pascanu, G Montufar, and Y Bengio. "On the number of inference regions of deep feed forward with piece-wise linear activations." In: *arXiv preprint arXiv:1312.6098* (2013).

[Pas+17] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. "Automatic differentiation in pytorch." In: (2017).

[PH22] Mary Phuong and Marcus Hutter. *Formal Algorithms for Transformers*. 2022. arXiv: 2207.09238 [cs.LG].

[RJS17] Alec Radford, Rafal Jozefowicz, and Ilya Sutskever. "Learning to generate reviews and discovering sentiment." In: *arXiv preprint arXiv:1704.01444* (2017).

[Rad+19] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. "Language models are unsupervised multitask learners." In: *OpenAI blog* 1.8 (2019), p. 9.

[RVL12] Tapani Raiko, Harri Valpola, and Yann Lecun. "Deep Learning Made Easier by Linear Transformations in Perceptrons." In: *Proceedings of the Fifteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Neil D. Lawrence and Mark Girolami. Vol. 22. Proceedings of Machine Learning Research. 2012, pp. 924–932.

[Ris86] Jorma Rissanen. "Stochastic complexity and modeling." In: *The annals of statistics* (1986), pp. 1080–1100.

[Ros58] Frank Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6 (1958), p. 386.

[Ros18] Richard J Rossi. *Mathematical statistics: an introduction to likelihood based inference*. John Wiley & Sons, 2018.

[Roz+23] Baptiste Rozière et al. *Code Llama: Open Foundation Models for Code*. 2023. arXiv: 2308.12950 [cs.CL].

[RHW+85] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. *Learning internal representations by error propagation*. 1985.

[Sch+19] Imanol Schlag, Paul Smolensky, Roland Fernandez, Nebojsa Jojic, Jürgen Schmidhuber, and Jianfeng Gao. "Enhancing the transformer with explicit relational encoding for math problem solving." In: *arXiv preprint arXiv:1910.06611* (2019).

[Sch92] Jürgen Schmidhuber. "Learning to control fast-weight memories: An alternative to dynamic recurrent networks." In: *Neural Computation* 4.1 (1992), pp. 131–139.

[Sch93] Jürgen Schmidhuber. "Reducing the ratio between learning complexity and number of time varying variables in fully recurrent nets." In: *ICANN'93: Proceedings of the International Conference on Artificial Neural Networks Amsterdam, The Netherlands 13–16 September 1993 3*. Springer. 1993, pp. 460–463.

[Şen+18] Lütfi Kerem Şenel, Ihsan Utlu, Veysel Yücesoy, Aykut Koc, and Tolga Cukur. "Semantic structure and interpretability of word embeddings." In: *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 26.10 (2018), pp. 1769–1779.

[Sha21] Atefeh Shahroudnejad. *A Survey on Understanding, Visualizations, and Explanation of Deep Neural Networks*. 2021. arXiv: 2102.01792 [cs.LG].

[SSS14] Shai Ben-David Shai Shalev-Shwartz. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014.

[Shi+99] Yusuxke Shibata, Takuya Kida, Shuichi Fukamachi, Masayuki Takeda, Ayumi Shinohara, Takeshi Shinohara, and Setsuo Arikawa. "Byte Pair encoding: A text compression scheme that accelerates pattern matching." In: (1999).

[SGK17] Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. "Learning important features through propagating activation differences." In: *International conference on machine learning*. PMLR. 2017, pp. 3145–3153.

[Sol64] Ray J Solomonoff. "A formal theory of inductive inference. Part I." In: *Information and control* 7.1 (1964), pp. 1–22.

[Spi18] Michael Spivak. *Calculus on manifolds: a modern approach to classical theorems of advanced calculus*. CRC press, 2018.

[SGS15] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. *Training Very Deep Networks*. 2015. arXiv: 1507.06228 [cs.LG].

[Sut23] Ilya Sutskever. *No Priors Ep. 39 | With OpenAI Co-Founder & Chief Scientist Ilya Sutskever*. https://www.youtube.com/watch?v=Ft0gTO2K85A. [Online; accessed 28.11.2023]. 2023.

[SVL14] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. *Sequence to Sequence Learning with Neural Networks*. 2014. arXiv: 1409.3215 [cs.CL].

[Sze+15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. "Going deeper with convolutions." In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 1–9.

[UVL17] Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. *Instance Normalization: The Missing Ingredient for Fast Stylization*. 2017. arXiv: 1607.08022 [cs.CV].

[Vas+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. *Attention Is All You Need*. 2017. arXiv: 1706.03762 [cs.CL].

[VWB16] Andreas Veit, Michael J Wilber, and Serge Belongie. "Residual networks behave like ensembles of relatively shallow networks." In: *Advances in neural information processing systems* 29 (2016).

[Vir+20] Pauli Virtanen et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python." In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

[Wer74] Paul Werbos. "Beyond regression: New tools for prediction and analysis in the behavioral sciences." In: *PhD thesis, Committee on Applied Mathematics, Harvard University, Cambridge, MA* (1974).

[WH18] Yuxin Wu and Kaiming He. *Group Normalization*. 2018. arXiv: 1803.08494 [cs.CV].

[Xio+20] Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tieyan Liu. "On layer normalization in the transformer architecture." In: *International Conference on Machine Learning*. PMLR. 2020, pp. 10524–10533.

[Xu+19] Jingjing Xu, Xu Sun, Zhiyuan Zhang, Guangxiang Zhao, and Junyang Lin. *Understanding and Improving Layer Normalization*. 2019. arXiv: 1911.07013 [cs.LG].

[You+19] Kaichao You, Mingsheng Long, Jianmin Wang, and Michael I. Jordan. *How Does Learning Rate Decay Help Modern Neural Networks?* 2019. arXiv: 1908.01878 [cs.LG].

[Zha+22] Yuxue Zhao, Bo Hu, Ying Wang, Xiaomeng Yin, Yuanyuan Jiang, and Xiuli Zhu. "Identification of gastric cancer with convolutional neural networks: a systematic review." In: *Multimedia Tools and Applications* 81.8 (2022), pp. 11717–11736.

[Zho+21] Haoyi Zhou, Shanghang Zhang, Jieqi Peng, Shuai Zhang, Jianxin Li, Hui Xiong, and Wancai Zhang. "Informer: Beyond efficient transformer for long sequence time-series forecasting." In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 35. 12. 2021, pp. 11106–11115.

[Zho+18] Wenda Zhou, Victor Veitch, Morgane Austern, Ryan P Adams, and Peter Orbanz. "Compressibility and generalization in large-scale deep learning." In: *arXiv preprint arXiv:1804.05862* 2 (2018).