Applications of Diversity and the Self-Attention Mechanism in Neural Networks

Berg, Axel

2022

Document Version:
Publisher's PDF, also known as Version of record

Applications of Diversity and the Self-Attention Mechanism in Neural Networks

by Axel Berg

LUND
UNIVERSITY

Centre for Mathematical Sciences
Lund University
Box 118
SE-221 00 Lund
Sweden

www.maths.lth.se

Licentiate Thesis in Mathematical Sciences 2022:1
ISSN: 1404-0034

ISBN: 978-91-8039-152-8 (print)
ISBN: 978-91-8039-151-1 (electronic)
LUTFMA-2045-2022

Printed in Sweden by Media-Tryck, Lund University, Lund 2022

Abstract

This thesis covers three contributions in applications of neural networks. The first is related to diversity and ensemble learning, while the other two cover novel applications of the self-attention mechanism.

An important aspect of training a neural network is the choice of objective function. Regression via Classification (RvC) is often used to tackle problems in deep learning where the target variable is continuous, but standard regression objectives fail to capture the underlying distance metric of the domain. This can result in better performance of the trained model, but the optimal choice of discrete classes used in RvC is not well understood. In Paper I , we introduce the concept of label diversity by generalizing the RvC method. By exploiting the fact that labels can be generated in arbitrary ways for continuous and ordinal target variables, we show that using multiple labels can improve the prediction accuracy of a neural network compared to using a single label and provide theoretical justification from ensemble theory. We apply our method to several tasks in computer vision and show increased performance compared to regression and RvC baselines.

The performance of a neural network is also influenced by the choice of network architecture, and in the design process it is important to consider the domain of the inputs and its symmetries. Graph neural networks (GNNs) is the family of networks that operates on graphs, where information is propagated between the graph nodes using for example self-attention. However, self-attention can be used for other data domains as well if the inputs can be converted into graphs, which is not always trivial. In Paper II , we do this for audio by using a complete graph over audio features extracted from different time slots. We apply this technique to the task of keyword spotting and show that a neural network solely based on self-attention is more accurate than previously considered architectures.

Finally, in Paper III we apply attention-based learning to point cloud processing, where the permutation symmetry must be preserved. In order to make the self-attention mechanism both more efficient and more expressive, we propose a hierarchical approach that allows individual points to interact on both a local and global scale. By extensive experiments on several benchmarks, we show that this approach improves the descriptiveness of the learned features, while simultaneously reducing the computational complexity compared to an architecture that applies self-attention naively on all input points.

List of Publications

The contents of this thesis is based on the following publications:

I **Deep Ordinal Regression with Label Diversity**

Axel Berg, Magnus Oskarsson, Mark O'Connor
2020 25th International Conference on Pattern Recognition (ICPR), 2021, pp. 2740-2747, doi: 10.1109/ICPR48806.2021.9412608.

II **Keyword Transformer: A Self-Attention Model for Keyword Spotting**

Axel Berg, Mark O'Connor, Miguel Tairum Cruz
Proc. Interspeech 2021, 4249-4253, doi: 10.21437/Interspeech.2021-1286

III **Points to Patches: Enabling the Use of Self-Attention for 3D Shape Recognition**

Axel Berg, Magnus Oskarsson, Mark O'Connor
Manuscript, Lund University, 2022

Author's Contributions

I I came up with the idea after some help from the other authors, developed the theory and implemented the experiments. The paper was written by me, with input from the other authors.

II Mark came up with the idea of applying Transformers to keyword spotting and I implemented the model. I implemented most of the experiments, except for knowledge distillation, which was done by Mark, and latency measurements, which was done by Miguel. The paper was written jointly by the three authors.

III I came up with the idea after some discussions with the other authors. The implementation and all experiments were done by me. The paper was written by me, with input from the other authors.

Acknowledgements

I would like to express my gratitude towards my main supervisors Magnus Oskarsson and Mark O'Connor for guiding me through the research process. It has been a very rewarding experience so far and your advice has been invaluable throughout the process.

I would also like to thank my co-supervisor Kalle Åström, the Arm ML Research team, as well as my colleagues at Matematikcentrum and the WASP community for interesting discussions. Finally, I would like to thank my wife Anna for her support and encouragement.

Funding

This work was financially supported by Arm and by the Wallenberg AI, Autonomous Systems and Software Program (WASP), funded by the Knut and Alice Wallenberg Foundation.

.

.

List of Abbreviations

k-NN k-nearest neighbour

ADAM Adaptive moment estimation

ASR Automatic speech recognition

BN Batch normalization

CE Cross entropy

CNN Convolutional neural network

DCT Discrete cosine transform

FLOPS Floating point operations per second

FMR Feature matching recall

GNN Graph neural network

LN Layer normalization

MAE Mean average error

MAP Maximum a posteriori probability

MFCC Mel-frequency cepstrum coefficients

mIoU mean intersection-over-union

ML Maximum likelihood

MLP Multi-layer perceptron

MSA Multi-head self-attention

MSE Mean squared error

RANSAC Random sample consensus

RNN Recurrent neural network

RvC Regression via classification

SA Self-attention

SGD Stochastic gradient descent

SLAM Simultaneous localization and mapping

Contents

Introduction

1 Background

Feature extraction is an important step in pattern recognition that deals with finding useful representations of data, for example images or audio, which can be used for downstream tasks. Sometimes it is possible to design hand-crafted feature extractors based on human intuition, but in recent years they have fallen out of favor and been replaced by machine learning models that learn to extract features based on patterns in large data collections. In recent years, the use of neural networks as feature extractors has accelerated progress in many fields, perhaps most notably in computer vision [24]. In section 2, we provide a brief introduction to some important concepts in deep learning that are necessary for understanding the rest of the material in this thesis.

When using a neural network for a specific task, there are several factors that influence the final performance of the network, including the size and quality of the data set used for training the network, the optimization procedure and choice of objective function, and the architecture of the neural network. In general, these factors cannot be disentangled, because changing one of them can influence how one ought to decide on the others. Although perhaps unrealistic, research in this area therefore often builds upon the assumption that all but one factor are fixed. Furthermore, the data set, which is perhaps the most important factor, cannot be altered or extended in many scenarios. Also, neural networks generally converge when optimized using simple methods based on gradient descent, which makes the choice of optimization procedure less relevant for the final network performance [9]. This leaves the researcher with two important design choices: the objective function and the network architecture.

This thesis covers different aspects of both of these problems. In particular, we discuss how different representations of the inputs and outputs of the network leads to different design choices. For example, consider the problem of age estimation, where the input $\mathbf{x}$ is an image of a human face and the output y is the age of the person in the image. Firstly, the choice of objective function will depend on how the age represented. Perhaps the most

natural choice is to let $y \in \mathbb{R}^+$, i.e. the age is a positive real number. However, in some scenarios the labeled examples in the training data only contains the year of birth of each person. Therefore, another natural choice would be to use a set of positive integers, e.g. $y \in \{0, 1, ..., 100\}$. The difference between the choice of output representation is subtle, but it has consequences for both the objective function and the network design. In terms of the objective function, the first choice leads to a regression problem where the network has a single output that directly predicts the age of the person in the image. The second choice leads to a classification problem, where the network needs to produce 101 outputs, and the k:th output predicts a probability that the age of the person is equal to k. Is it possible to say which of these two choices is better in terms of the final predictive performance of the network? In this Section 3 we explore the pros and cons of the different output representations, which leads us to presenting a novel objective function that exploits the ambiguity in how the outputs are represented.

Likewise, the network architecture will depend on how the image is represented. Typically, a digital image $\mathbf{x}$ with resolution $h \cdot w$ is represented as a set of pixels on a 2D grid, where each pixels contains three color channels: red, green and blue (RGB). Therefore, a natural image representation is $\mathbf{x} \in \mathbb{R}^{h \times w \times 3}$. This allows for the use of grid-based operations, such as two dimensional convolutions, when extracting image features. However, this is not the only possible representation, and recent works [11, 22] suggest that splitting the image into a set of small patches and processing it using graph-based learning methods can be more effective in terms of extracting descriptive features. The same reasoning can be applied to other types of data, for example audio, which is often processed using convolutions on a 2D spectrogram.

The use of graphs to represent data is closely related to permutation symmetry, since a graph has no intrinsic ordering of its nodes. Another type of data which is commonly modeled using graphs is unordered 3D point clouds from scans of real-world environments. Fast and accurate processing of point clouds is an important step in many navigation applications, such as advanced driver-assistance systems (ADAS) and simultaneous localization and mapping (SLAM). In Section 4 we outline the concept of permutation symmetry, how it relates to modern graph neural network architectures and discuss some of its applications. Finally, in section 5, we conclude the introduction by outlining the contents of the included publications.

2 Artificial Neural Networks and Deep Learning

An artificial neural network consists of a set of neurons, with corresponding weights and activation functions, that form a computational graph. A network $f_{\boldsymbol{\theta}} : \mathcal{X} \to \mathcal{Y}$, parameterized by a set of parameters $\boldsymbol{\theta}$, defines a mapping from the input space $\mathcal{X}$ to the output

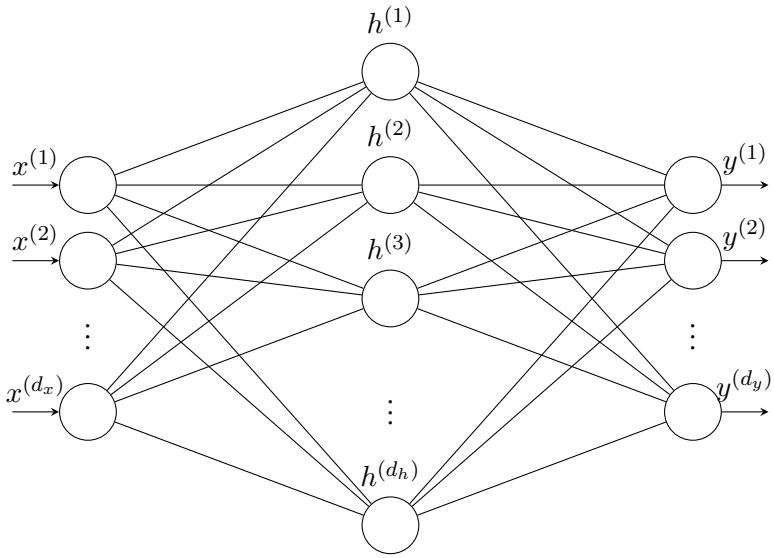

Figure 1: An MLP with a single hidden layer. The inputs $x^{(1)}, ..., x^{(d_x)}$ are first propagated to the hidden layer, where activations $h^{(1)}, ..., h^{(d_h)}$ are computed. These are then propagated to the final layer, where the output $y^{(1)}, ..., y^{(d_y)}$ of the neural network is obtained.

space $\mathcal{Y}$. The simplest form of neural network is the perceptron [20], which was originally inspired by neurons in the brain, that maps an input vector $\mathbf{x} = [x^{(1)}, ..., x^{(d)}]^T$ to an output y, using a weighed sum

$$y = \sigma(\mathbf{x}^T\mathbf{w} + b). \tag{1}$$

Here σ is a non-linearity, also known as the activation function. The perceptron is parameterized by the weight vector $\mathbf{w} = [w^{(1)}, ..., w^{(d)}]^T$ and the bias b, which can be modified in order for the perceptron to compute different linear functions.

By combining multiple layers of perceptrons, we get a directed computational graph which is referred to as a multi-layer perceptron (MLP), as shown in Figure 1. The neurons in-between the input and the output layer are "hidden" layers and their outputs store intermediate values that are propagated forward through the network. By increasing the number of hidden layers in the network, we arrive at what is commonly referred to as deep neural networks, although there is no strict definition on where the line between shallow and deep networks goes.

Under certain conditions, MLPs are universal function approximators, in the sense that they can approximate any continuous function with arbitrary precision. In theory, a single hidden layer with sufficient width is enough to achieve this property [3], but in practice deeper networks have shown to generalize better to unseen data [8].

2.1 Supervised Learning

When performing function approximation using a neural network, a learning rule is necessary to update the parameters of the network. In the context of supervised learning, this is done by minimizing a loss function, which depends on the problem type. More specifically, given a set of input and output pairs $\mathcal{S} = \{(\mathbf{x}_n, t_n)\}_{n=1}^N$, we want to update the parameters of our network such that we minimize some objective function $L : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}^+$, which often is referred to as the loss function. Under the assumption that the data samples are independent and identically distributed (i.i.d.), we seek to optimize the parameters $\boldsymbol{\theta}$ such that the likelihood $\log p_{\boldsymbol{\theta}}(t_n|\mathbf{x}_n)$ of the observed data is maximized. This is equivalent to minimizing the negative log-likelihood:

$$\boldsymbol{\theta}_{\text{ML}} = \arg\max_{\boldsymbol{\theta}} \prod_{n=1}^{N} p_{\boldsymbol{\theta}}(t_n|\mathbf{x}_n) = \arg\min_{\boldsymbol{\theta}} -\sum_{n=1}^{N} \log p_{\boldsymbol{\theta}}(t_n|\mathbf{x}_n). \tag{2}$$

This is known as the maximum likelihood (ML) estimate and it will take various forms depending on how we choose to model the data distribution. For example, assume that the underlying process generating the data can be modeled as $t_n = f_{\boldsymbol{\theta}}(\mathbf{x}_n) + \epsilon_n$, where $\epsilon_n \sim \mathcal{N}(0, \sigma^2)$ are i.i.d. random variables sampled from a Gaussian distribution and $f_{\boldsymbol{\theta}}$ is our neural network model. Our ML estimate in (2) then simplifies to

$$\boldsymbol{\theta}_{\text{ML}} = \arg\min_{\boldsymbol{\theta}} -\sum_{n=1}^{N} \log \frac{1}{\sqrt{2\pi\sigma^2}} \exp(-\frac{(t_n - f_{\boldsymbol{\theta}}(\mathbf{x}_n))^2}{2\sigma^2}) \tag{3}$$

$$= \arg\min_{\boldsymbol{\theta}} \frac{N}{2} \log 2\pi\sigma^2 + \sum_{n=1}^{N} \frac{1}{2\sigma^2}(t_n - f_{\boldsymbol{\theta}}(\mathbf{x}_n))^2. \tag{4}$$

Ignoring constants, we find that maximizing the likelihood corresponds to minimizing the squared error between the network predictions and the ground truth data labels. In this case it is therefore natural to use the mean squared error as loss function:

$$L_{\text{MSE}}(\mathbf{x}, t, \boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^{N} (t_n - f_{\boldsymbol{\theta}}(\mathbf{x}_n))^2. \tag{5}$$

When choosing the neural network architecture for this problem, which is a form of non-linear regression, the last layer of the network has a single output neuron such that the predictions $y_n = f_{\boldsymbol{\theta}}(\mathbf{x}_n) \in \mathbb{R}$. Furthermore, the regression problem can easily be extended to higher dimensions.

In other scenarios, networks are used for multi-way classification problems where the targets $t_n \in \{1, ..., K\}$ belong to a set of discrete categories. The network architecture then has to

be modified to use K output neurons, where each neuron models the probability $p_{\boldsymbol{\theta}}(k|\mathbf{x}_n)$ of $\mathbf{x}_n$ being an instance of class k. In order to generate a probability distribution over classes, the output of the last hidden layer is typically normalized using a softmax function

$$\text{softmax}(h)^{(k)} = \frac{e^{h^{(k)}}}{\sum_{j=1}^{K} e^{h^{(j)}}}. \tag{6}$$

This guarantees that the predicted probabilities are non-negative and sum to 1. While there are other functions with these properties, the softmax is most commonly used because it has several desirable properties, such as invariance to additive scalars. Furthermore, the softmax can be interpreted as a smooth differentiable approximation of the arg max function [8].

Although the performance of a classification model is usually evaluated using its accuracy, i.e. the fraction of correctly classified data samples, this objective is difficult to optimize in practice. Therefore it is more common to use a loss function that measures the similarity between the predicted distribution and the ground truth distribution $q(k|\mathbf{x}_n)$, such as the cross-entropy (CE), which for an individual data sample is defined as

$$H(q, p_{\boldsymbol{\theta}}) = -\sum_{k=1}^{K} q(k|\mathbf{x}_n) \log p_{\boldsymbol{\theta}}(k|\mathbf{x}_n). \tag{7}$$

Since the data points belong to a set of discrete classes, we can use a one-hot encoding to model the ground truth distribution:

$$q(k|\mathbf{x}_n) = \begin{cases} 1, & t_n = k \\ 0, & \text{otherwise.} \end{cases} \tag{8}$$

Note that the choice of $q(k|\mathbf{x}_n)$ need not be a one-hot distribution. For example, label smoothing [21], which is a form of network regularization, uses a smoothing parameter α that penalizes over-confident predictions

$$q(k|\mathbf{x}_n) = \begin{cases} 1 - \alpha, & t_n = k \\ \frac{\alpha}{K}, & \text{otherwise.} \end{cases} \tag{9}$$

For ordinal regression problems, where the classes can be ranked on an ordinal scale, there are other several other possible label encodings, which are further discussed in Section 3.

There exists an intimate connection between the CE loss and the ML estimation technique. Using the one-hot encoding in 8, we can calculate the average cross-entropy across the entire

data set as

$$L_{\text{CE}}(\mathbf{x}, t, \boldsymbol{\theta}) = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} q(k|\mathbf{x}_n)\log p_{\boldsymbol{\theta}}(k|\mathbf{x}_n) = \frac{1}{N}\sum_{n=1}^{N}\log p_{\boldsymbol{\theta}}(t_n|\mathbf{x}_n), \quad (10)$$

where $p_{\boldsymbol{\theta}}(t_n|\mathbf{x}_n)$ here should be interpreted as the likelihood of the parameters $\boldsymbol{\theta}$ given the observations $(\mathbf{x}_n, t_n)$. By comparing this with equation (2), we can conclude that under these assumptions, minimizing the average cross entropy is in fact equivalent to minimizing the average negative log-likelihood.

2.2 Training a Neural Network

Training a neural network is an optimization problem where the goal is to minimize the expected loss $L(\mathbf{x}, t, \boldsymbol{\theta})$ for the entire data set. Ideally, if the network is able to recognize patterns in the training data, and consequently minimize the loss function, it will be able to infer the same patterns on data not seen during training. An assumption is that the unseen data is drawn from the same underlying distribution as the training data. For example, if a neural network has been trained to classify images of cats and dogs, it will hopefully generalize to unseen examples of cats and dogs. However, if the training set only contains dogs of specific races, e.g. Labrador Retrievers, the classification performance might not generalize well to images of Chihuahuas.

Another common problem in neural network training is poor generalization due to memorization of training examples, also referred to as overfitting. This can occur when the parameter space is too large, which in practice often means that the network is too deep (has too many layers) or too wide (each layer has too many parameters). Therefore, an obvious way to regularize a network is to make it smaller, but in practice it has been shown that over-parameterized networks perform better even on simpler tasks [30, 16]. Instead, different regularization strategies are thus applied during training, which can be done for example by modifying the loss function to penalize memorization of training examples. For instance, weight decay (WD) can be applied by adding an extra term to the loss function which penalizes the l^2-norm of the parameters:

$$L_{\text{WD}}(\mathbf{x}, t, \boldsymbol{\theta}) = L(\mathbf{x}, t, \boldsymbol{\theta}) + \frac{\lambda}{2}\boldsymbol{\theta}^T\boldsymbol{\theta}. \quad (11)$$

Here, $\lambda > 0$ is a hyperparameter that determines the regularization strength. The intuition behind weight decay is that it encourages the optimizer to find an "simple" solution where the weights are close to 0. Furthermore, it can be shown that adding weight decay in the loss function is equivalent to setting a Gaussian prior on $\boldsymbol{\theta}$ [5].

Another common technique is to use data augmentation in order to artificially expand the training set by feeding the network multiple augmented copies of the same data samples. For example, when learning to classify images of dogs and cats, we can use random scaling, rotations and cropping of the images. This also forces the network to be approximately invariant to those transformations, since the labels are not being changed (a rotated cat is still a cat). In some cases, it may also be desirable to make the network *exactly* invariant by restricting the architecture of the network itself. This topic is further explored in Section 4.

In practice, neural networks are optimized using various forms of gradient descent. The optimization procedure consists of 1) estimating the gradient $\nabla_{\boldsymbol{\theta}} L$ of the loss function with respect to the network parameters and 2) updating the parameters in the negative direction of the gradient. In its simplest form, the learning rule is given by

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} L, \tag{12}$$

where η is the learning rate. The update is applied repeatedly until convergence is reached, which usually requires iterating over the training data multiple times. Since the neural network is essentially a computational graph, the gradient with respect to individual scalar parameter of the network can be found using backpropagation, which involves applying the chain rule recursively.

In practice, it is not computationally efficient to compute the expected gradient using all training samples, since only a subset is needed to get an estimate of the gradient. The most common learning rule is stochastic gradient descent (SGD), where for each update the gradient is estimated using a randomly sampled subset of the training data. More specifically, each update uses minibatch $\mathcal{B} \subset \mathcal{S}$ of training data and then estimates the gradient as

$$\nabla_{\boldsymbol{\theta}} \approx \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x}_n, y_n \in \mathcal{B}} \nabla_{\boldsymbol{\theta}} L(\mathbf{x}_n, y_n, \boldsymbol{\theta}). \tag{13}$$

In recent years, more advanced learning rules that use gradient momentum in order to adaptively adjust the learning rate, such as Adam [10], have been proposed in order to reach faster convergence. Together with the fast evolution of modern computer hardware, such as graphics (GPUs) and neural (NPUs) processing units, that allow for efficient acceleration of both training and inference on computational graphs, have made deep neural networks both a ubiquitous tool in the research community, but also feasible to deploy in many commercial applications. In this section we have only made a brief introduction to this topic and the reader is referred to [8] for a more rigorous treatment of basic deep learning concepts.

3 Diversity and Ordinal Regression

In this section, we will cover the notion of diversity in the context of statistical learning theory. Here, diversity refers to differences of properties within a collection of models, data sets or predictions. Hence, a diverse set of models can mean either models with different architectures or different values of their parameters. Likewise, a diverse data set contains a wide variety of samples that are not biased towards a particular setting. For example, a non-diverse data set with images of cats and dogs might only contain instances of particular breeds, or only images captured indoors, whereas a diverse data set should capture enough variety in both of these aspects. Obviously, it is not trivial to quantify diversity in this aspect, since the notion is inherently subjective and depends on the context. However, a diversity in predictions is easier to quantify, since we can measure the variance of the predictions. Hence a set of predictions can be considered diverse if they have a variance larger than zero.

An analogy for diverse predictions can be made with juries used for grading in sport competitions, for example gymnastics and figure skating, where the final score is assigned to each competitor by averaging the scores of the individual jurors. This mitigates the influence of biased jurors and is therefore considered more fair than a single juror, a phenomenon sometimes referred to as the wisdom of the crowd. Likewise, in countries where juries are used in judicial courts, a convicting verdict can only be reached if a large majority of jurors are in agreement. Furthermore, during the jury selection process, it is often considered desirable to have diversity within the jury with respect to e.g. income, gender and occupation, which is similar to the notion of model diversity.

In communication science, diversity can be exploited in order to mitigate the influence of random processes. For example, when a mobile phone has poor signal reception, the cell tower may transmit a repeated version of the signal, which can then be averaged in the receiver. The white noise generated in the receiver antenna will then tend to zero as the number of repetitions increase. In some applications, multiple receiver or (and) transmitter antennas are used as well, which increases the probability of at least one antenna pair having good reception [28].

Diversity in machine learning is closely related to ensemble learning, in which a set of model predictions are combined in order to improve predictive performance compared to a single model on a given task. An old saying goes “two heads are always better than one”, and as we shall prove in this section, the error of an ensemble average is always smaller than the average error of the individual ensemble members, as long as there is enough diversity among the ensemble members. This makes it possible to combine a set of weak models, with predictive performance only slightly better than random guessing, into a strong ensemble model with good predictive performance, a method known as boosting [7].

3.1 Regression Ensembles

For a given problem, we might have several models $f^m_{\boldsymbol{\theta}}$, $m = 1, ..., M$, that we wish to combine in order create an ensemble model. For a regression problem with predictions $y_m = f^m_{\boldsymbol{\theta}}(x)$, the simplest form of combination is to use a linear combination of the individual predictions

$$y_{ens} = \sum_{m=1}^{M} w_m y_m, \tag{14}$$

where set of weights $\{w_m\}_{m=1}^M$ are typically restricted to be non-negative and sum to 1, which is also referred to as a convex linear combination. The simplest form of linear combination is to assign equal weight to each individual predictor, i.e. $w_m = 1/M, \forall m$, but it is also possible to optimize them to minimize the prediction error on a validation data set [17].

Given that we have calculated an ensemble average of our predictions, we would like to understand the benefit from this operation. Fortunately, there exists an easy way to decompose the ensemble error, which allows us to quantify the error directly in terms of the errors of the individual predictors.

Theorem 3.1. The Ambiguity Decomposition (Krogh and Vedelsby [12]) Consider an ensemble of $m = 1, ..., M$ models and the convex linear ensemble average computed as in (14). The quadratic error between the ensemble average y_{ens} and the target t then satisfies

$$(y_{ens} - t)^2 = \sum_{m=1}^{M} w_m (y_m - t)^2 - \sum_{m=1}^{M} w_m (y_m - y_{ens})^2. \tag{15}$$

Proof: The result can easily be shown by manipulation of terms:

$$\sum_{m=1}^{M} w_m (y_m - t)^2 = \sum_{m=1}^{M} w_m (y_m - y_{ens} + y_{ens} - t)^2 \tag{16}$$

$$= \sum_{m=1}^{M} w_m \left[(y_m - y_{ens})^2 + (y_{ens} - t)^2 + 2(y_m - y_{ens})(y_{ens} - t) \right] \tag{17}$$

$$= (y_{ens} - t)^2 + \sum_{m=1}^{M} w_m (y_m - y_{ens})^2, \tag{18}$$

where we have used the fact that $y_{ens} = \sum_m w_m y_m$ and $\sum_m w_m = 1$. The final result is obtained by rearranging the terms. □

The ambiguity decomposition states that the error of the ensemble average is less than or equal to the average error of the individual predictions of the ensemble members. In order for the ensemble error to be small, the first term in (15) should be minimized, which requires that the individual predictions are accurate. However, the second term, which grows with increased variance within the ensemble predictions, should be large. Therefore there is a trade-off to be made between accurate and diverse ensemble members.

3.2 Regression via Classification

For specific problems, using direct regression might be undesirable for several reasons. Consider for example the problem of pose estimation, where the target variable $t \in [0, 2\pi)$ is an angle of rotation. If we use the standard MSE loss function as in (5), it will suffer from discontinuity at $t = k2\pi$, $k \in \mathbb{Z}$. For example, if the target is $t = 2\pi - \epsilon$ for some small value of ϵ, and the model predicts $y = \epsilon$, the error modulo 2π is 2ϵ. However, the squared error is $(2\pi - 2\epsilon)^2$, so using the MSE loss will not be suitable for this problem.

We could try to use another loss function, for example $L = \cos(y - t)$, but since this loss has several local minima, it would not be suitable for optimization. This could be solved for example by letting the model predict a pair of coordinates on the unit circle, instead of directly predicting the pose angle, but this would require a normalization of the outputs that makes optimization difficult. Here we will instead focus another method known as regression via classification (RvC).

The main idea of RvC is to consider the target as a categorical variable, which makes it possible to use classification methods for predictions. However, it is not always obvious how the categorical classes ought to be defined for a given problem. The simplest solution is to create bins of equal width that span the entire domain of the target. In the case of pose estimation where the target variable is an angle, we can for instance divide it into K equally wide bins, forming a set of categorical variables $\{c_k\}_{k=1}^K$, where each class corresponds to an interval, such that $c_k = [\frac{k-1}{K}2\pi, \frac{k}{K}2\pi)$. Now, we can define a mapping from the continuous targets to the K classes by simply assigning class k to all targets that belong to the interval c_k, as shown in Figure 2.

When selecting the number of intervals K that spans the target domain, there is a trade-off to be made between the discretization error and the number of training examples in each class. For example, if we assume that our training data consists of N data points we can let $K = N$ and have one training example per class. This will make it difficult for the model to generalize to unseen data points, since it will not learn to group nearby angles in

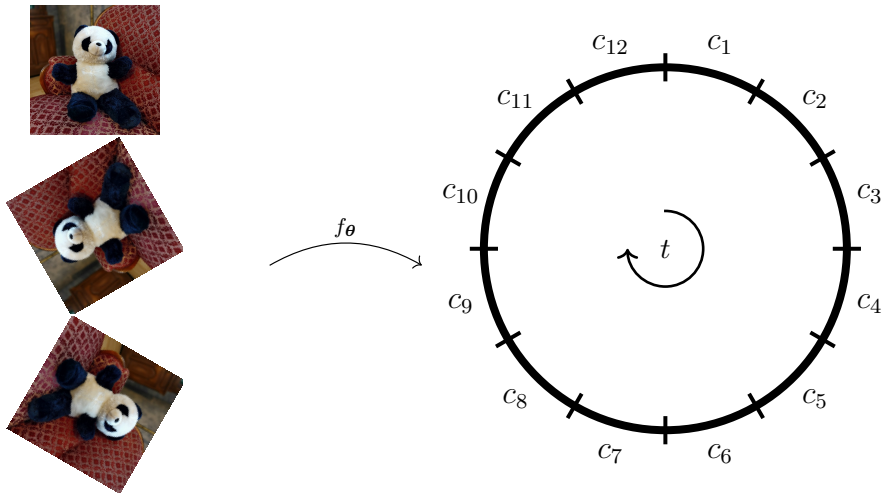

Figure 2: An example of how to perform RvC for image rotation prediction. Here, each continuous angle $t \in [0, 2\pi)$ is mapped to one of $K = 12$ categorical labels using intervals of equal width on the unit circle. The model then tries to classify each image in the correct bin, instead of predicting the angle directly.

the same category. On the other extreme end, we could let $K = 2$, such that $c_1 = [0, \pi)$ and $c_2 = [\pi, 2\pi)$, which corresponds to a binary classification problem where the pose is classified into two segments of the unit circle. Obviously, this is not a good choice, because even if the classifier can achieve high accuracy, we want to be able to infer poses with higher precision than this. Therefore we can conclude that the optimal number of bins lies somewhere in between these extremes, although there is no general rule that can be inferred on what the optimal value for K ought to be.

When training the model, we assign each training data point to one of the classes and create labels using e.g. a one-hot distribution

$$q(c_k) = \begin{cases} 1, & t \in c_k \\ 0, & \text{otherwise.} \end{cases} \tag{19}$$

The model is trained to match the labels by predicting a probability distribution $p(t_n \in c_k|\mathbf{x}_n)$ that matches the labels. For brevity, we will from here forth denote the predicted probability simply as $p(c_k|\mathbf{x})$.

Model optimization can be done using the same procedure as for an ordinary classification problem, using e.g. the cross-entropy loss function

$$L = -\sum_{k=1}^{K} q(c_k) \log p(c_k|\mathbf{x}). \tag{20}$$

The RvC approach might seem counter intuitive at first, but it effectively solves the problem of discontinuous loss functions, since predictions on either side of the cutoff angle $t = 2\pi$ will be penalized equally. However, the price that we pay for using a categorical label representation is that the loss function is now agnostic to the magnitude of the error, because it does not take into account which classes are nearby. For example, using the categorical labels in Figure 2, if the true angle lies in c_1, the loss will be the same if the model predicts c_2 or c_7, although c_2 is clearly a better prediction.

From this it becomes obvious that the RvC method has thrown away useful information about the labels, namely 1) the ordinal relationship between the classes and 2) the distance metric used to compute the prediction errors, which in this case is the shortest path along the unit circle. However, there are ways to exploit this information even in the context of RvC. One solution, proposed by Diaz et al. [4], is to use a soft label representation

$$p(c_k|\mathbf{x}) = \frac{e^{-\phi(\gamma(c_k),t)}}{\sum_{i=1}^{K} e^{-\phi(\gamma(c_k),t)}}, \tag{21}$$

where ϕ is a distance metric and $\gamma(c_k)$ is the midpoint of the interval. In other words, the probability assigned to each class is inversely proportional to the distance between the class and the true class. In practice, this will penalize nearby class predictions less than far away predictions according to the chosen distance metric.

3.3 Ordinal Regression

Now let us consider a problem domain where there doesn't exist a natural distance metric. Consider for example the problem of ranking, where the objective is to classify objects into ordinal categories $t \in \{1, .., K\}$ and the classes can be ordered, but there doesn't exist a well-defined distance between classes. An example of such a problem would be to predict survey responses on a Likert scale, i.e. from "strongly disagree" to "strongly agree", based on some features of the respondent. This setting is referred to as ordinal regression and several methods have been proposed for solving it.

Frank et al. [6] proposed to treat K-class ranking problem as $K - 1$ separate binary classification problems by noting that instead of predicting "what is the rank of x?", asking "is the rank of x greater than k", for $k = 1, ..., K - 1$. In a machine learning setting, a model can then be trained to predict $p(t > k|x)$ for all relevant values of k, and the ranking

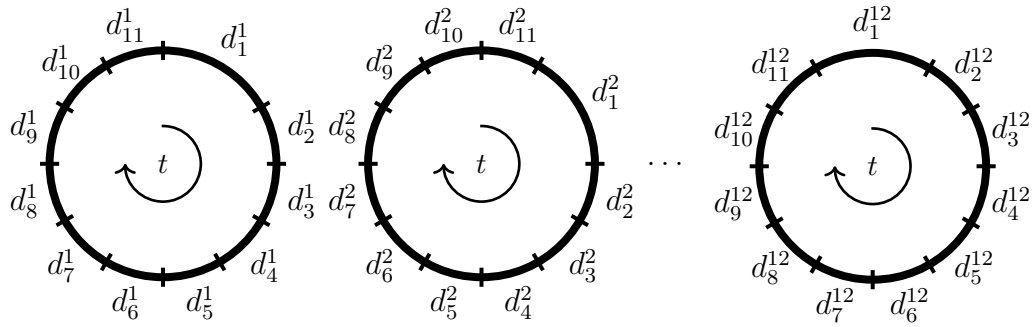

Figure 3: An example of how to create diverse sets of intervals by combining nearby classes. In this example, there are $M = 12$ different sets of labels, each containing $L = 11$ intervals that span the unit circle.

probabilities can then be recovered by noting that

$$\begin{aligned} p(t = 1|\mathbf{x}) &= 1 - p(t > 1|\mathbf{x}), \\ p(t = k|\mathbf{x}) &= p(t > k|\mathbf{x}) - p(t > k - 1|\mathbf{x}), \quad k = 2, ..., K - 1 \\ p(t = K|\mathbf{x}) &= p(t > K - 1|\mathbf{x}). \end{aligned} \tag{22}$$

Although this method is simple, it does not automatically guarantee that the predictions are rank-consistent, i.e. that $p(t > 1|\mathbf{x}) \geq p(t > 2|\mathbf{x}) \geq ... \geq p(t > K - 1|\mathbf{x})$, but methods have later been developed to enforce rank-consistency [14]. Nevertheless, training a model by summing the individual losses of each binary classifier will result in an overall loss that penalizes errors based on their magnitude on the ordinal scale.

3.4 Label Diversity

In Paper 1 , we consider an alternative approach to RvC and ordinal regression that exploits diversity, but in the labels rather than in the data. This approach is inspired by the fact that there are many possible ways to create categorical labels from continuous targets and that there is no general method for knowing a priori which categorical representation will be most suitable for a particular task. Going back to the example of pose estimation in Figure 2, we note that there are several arbitrary design choices that have to be made dividing the unit circle into intervals. For instance, how do we choose the number of intervals K? Should the intervals have equal length and should there be overlap between them?

From ensemble theory we know that combining different model predictions, where each model has been trained on a different label representation, will yield a smaller prediction error that the average error from the individual models. Therefore, without knowing which representation is most suitable, we can create a diverse set of representations and use an

ensemble of models to combine the predictions on the different representations into a final predictions. For example, we could train M different models using different values of K, and combine the the model predictions using an ensemble average.

The method of label diversity extends ordinary multi-class classification and ordinal regression problems, since different labels can be grouped in arbitrary ways. For example, a K-way classification problem can be expanded to $M = K$ different L-way classification problems, where $L = K - R + 1$ by grouping R consecutive classes

$$\begin{aligned} d_1^1 &= c_1 \cup c_2 \cup ... \cup c_R, d_2^1 = c_{R+1}, ..., d_L^1 = c_K, \\ d_1^2 &= c_2 \cup c_3 \cup ... \cup c_{R+1}, d_2^2 = c_{R+2}, ..., d_L^2 = c_1, \\ &\vdots \\ d_1^K &= c_K \cup c_1 \cup ... \cup c_{R-1}, d_2^K = c_R, ..., d_L^K = c_{K-1}. \end{aligned} \tag{23}$$

For example, by combining $R = 2$ nearby classes in our discretization of the unit circle, we can create $M = 12$ different 11-way classification problems, as shown in Figure 3. Each individual predictor can then be trained to output a probability $p_m(d_l^m|\mathbf{x})$, using labels $q(d_l^m|\mathbf{x})$ obtained as in (19). Note that the way labels are combined in (23) is simply an illustrative example, since in general we don't need to restrict L to be the same for each of the M classifiers. In other words, each circle in Figure 3 could be divided into a different number of intervals.

Before combining the individual predictions, we first need marginalize over d_l^m to recover distributions $p_m(c_k|\mathbf{x})$ over the original class intervals c_k for each individual predictor m. We note that $p(c_k|d_l^m, \mathbf{x})$ only depends on the overlap between c_k and d_l^m. For example, using our sets of labels in Figure 2 and 3, we can infer that $p_1(c_1|\mathbf{x}) = \frac{1}{2} \cdot p_1(d_1^1|\mathbf{x})$. More generally we have that

$$p(c_k|d_l^m, \mathbf{x}) = \frac{||d_l^m \cap c_k||}{||d_m^l||} = \begin{cases} \frac{1}{R}, & d_l^m \cap c_k \neq \emptyset \\ 0, & \text{otherwise.} \end{cases} \tag{24}$$

This allows us to calculate the posterior likelihoods as

$$p_m(c_k|\mathbf{x}) = \sum_{l=1}^{L} p(c_k|d_l^m, \mathbf{x}) p_m(d_l^m|\mathbf{x}), \quad m = 1, ..., M. \tag{25}$$

Now we combine the predicted likelihoods of the ensemble using a convex combination:

$$p(c_k|\mathbf{x}) = \sum_{m=1}^{M} w_m p_m(c_k|\mathbf{x}). \tag{26}$$

If each classifier is trained by minimizing the cross-entropy loss, then the sum of the individual losses will incorporate the ordinal ranking of the targets. This can be seen by noting that nearby angles will fall into the same intervals for some classifiers, and different intervals for others. For example, a pair very small angles ϵ and $-\epsilon$ will be mapped to d_{11}^2 and d_{10}^2 respectively, but they will both be mapped to d_1^{12}. Hence the total loss will incorporate the fact that these angles are close, but not equal, and therefore method of label diversity can be regarded as a form of ordinal regression.

For regression problems where the target is continuous, the method of label diversity can be generalized even more. For example, we can consider generating M sets of intervals $\{\mathcal{D}^m\}_{m=1}^M$ where $\mathcal{D}^m = \{d_l^m\}_{l=1}^{L_m}$ are arbitrary discretizations of the unit circle, with the restriction that the intervals in each set are non-overlapping. For any particular angle, we can then directly map it to an interval in each set $\mathcal{D}^m$. Since this mapping will obviously not be injective, we will still have to deal with the discretization error.

There are several ways that the predicted likelihoods can be used to form an estimate y_m of the angle. One possibility is to use the midpoint $\gamma(d_l^m)$ of the interval with the highest likelihood, i.e. $y_m = \gamma(d_{l*}^m)$, where $l^* = \arg\max_l p(d_l^m|\mathbf{x})$. Another option is to use the expected value as $y_m = \sum_l \gamma(d_l^m) p(d_l^m|\mathbf{x})$. The predictions of the ensemble members can the be combined using a convex combination as in (14). From the ambiguity decomposition, we can then establish that the ensemble error will be smaller than the average error of the ensemble members, since a non-zero variance is induced by the different discretizations. If the discretization errors are small, but independent, then the ensemble average will mitigate the effects of discretization. Likewise, there will be a trade-off between the discretization error of each member and the number of members required in the ensemble for good accuracy.

There remains several questions on how to implement label diversity in practice, as well as the extent to which this is useful. We refer to Paper I for examples of applying label diversity on real problems and performance comparisons with standard regression and RvC.

4 Permutation Symmetry and the Self-Attention Mechanism

An important aspect of neural network architecture design is to incorporate geometric priors by making the networks either invariant (or equivariant) to certain transformations. As

was briefly mentioned in Section 2.2, this can be regarded as a form of regularization, since it restricts the search space of the optimization to be limited to a smaller set of functions that obey the imposed restrictions. In practice, this can be done by designing the computational graph of the network such that the invariance is satisfied. For example, for a particular problem, such as classifying cats and dogs, where we have two augmented versions of the same image $\mathbf{x}$ and $\tilde{\mathbf{x}}$, we can strive to design our network such that $f_{\boldsymbol{\theta}}(\mathbf{x}) = f_{\boldsymbol{\theta}}(\tilde{\mathbf{x}})$, regardless of the particular value of $\boldsymbol{\theta}$. In general, we say that a transformation that does not change the underlying property of the object is a symmetry of that object. For example, in the context of image classification, a rotated and translated cat, is still a cat.

In order to formalize the notion of invariant neural networks, it is useful to borrow some concepts from group theory. Here we will briefly state some useful definitions and we refer to [1] for a more in-depth treatment of the relationship between group theory and deep learning. Given a symmetry group $\mathfrak{G}$ with group actions $\mathfrak{g} \in \mathfrak{G}$, we define the corresponding group representations as $\rho : \mathfrak{G} \to \mathrm{GL}(\mathcal{X})$, where $\mathrm{GL}(\mathcal{X})$ is the general linear group consisting of invertible matrices of a particular degree. We can now make the following definition:

Definition 4.1. A function $f : \mathcal{X} \to \mathcal{Y}$ is invariant with respect to $\mathfrak{G}$ if $f(\rho(\mathfrak{g})x) = f(x)$ for all $x \in \mathcal{X}$ and $\mathfrak{g} \in \mathfrak{G}$. Similarly, if $\mathcal{X} = \mathcal{Y}$, f is said to be equivariant with respect to $\mathfrak{G}$ if $f(\rho(\mathfrak{g})x) = \rho(\mathfrak{g})f(x)$.

For example, consider the simple case where $\mathcal{X} = \mathbb{R}^N$ and $\mathfrak{G}$ is the group of linear scaling. The group actions $\mathfrak{g}$ are then different scalings of the input vector, and the group representations $\rho(\mathfrak{g})$ are scalar matrices $c\mathbf{I}$, where $c \in \mathbb{R}$ and $\mathbf{I}$ is the $N \times N$ identity matrix. A scale-invariant function should then satisfy $f(c\mathbf{x}) = f(\mathbf{x})$ for all $\mathbf{x} \in \mathbb{R}^N$. This can be achieved, for example, by letting $f(\mathbf{x}) = g(\mathbf{x}/||\mathbf{x}||)$ for some function g.

A common method used for designing neural networks that are invariant with respect to symmetry groups is to use compositions of invariant and equivariant layers, which can be realized by modifying the computational graph. For example, if a neural network can be decomposed into two layers as $f_{\boldsymbol{\theta}} = f^{\text{inv}}_{\boldsymbol{\theta}_1} \circ f^{\text{equiv}}_{\boldsymbol{\theta}_2}$, where $f^{\text{inv}}_{\boldsymbol{\theta}_1}$ is $\mathfrak{G}$-invariant and $f^{\text{equiv}}_{\boldsymbol{\theta}_2}$ is $\mathfrak{G}$-equivariant, then obviously $f_{\boldsymbol{\theta}}$ is going to be $\mathfrak{G}$-invariant. In general, if we can decompose a deep neural network containing D layers as $f_{\boldsymbol{\theta}} = f^{D}_{\boldsymbol{\theta}_D} \circ ... \circ f^{1}_{\boldsymbol{\theta}_1}$, where layers $1, ..., D-1$ are equivariant, then the entire network will be invariant (equivariant) if layer D is invariant (equivariant).

Going back to the MLP in Figure 1, we note that the function computed by this network will in general not exhibit any type of invariance, which has inspired the development of more advanced network architecture. In particular, convolutional neural networks (CNNs) have become popular in computer vision due to their inherent translation equivariance. In the next section we will take a closer look on graph neural networks and their relation to permutation symmetry.

4.1 Learning on Sets

We will now investigate how group invariances can be applied to design network operating on unordered sets and graphs. Let us first consider the problem of function approximation on sets consisting of vector-valued elements $\{\mathbf{x}_u\}_{u=1}^N$. (Here, the subscript u denotes a vector-valued set element of a single data point, not the u:th member of the data set as in the previous sections). Without loss of generality, we can stack the set elements in a feature matrix $\mathbf{X} = [\mathbf{x}_1, ..., \mathbf{x}_N]^T$ where each row corresponds to an element of the set, but the ordering of the rows is arbitrary. The task of the network is then to output either a global feature of the entire set or element-wise features, or both. Learning on sets has several interesting applications, including 3D shape recognition from point clouds and text retrieval [29]. In contrast to pixels on a grid, the set elements need not have a well-defined ordering, and therefore our function approximator should yield the same prediction regardless of how the rows of $\mathbf{X}$ are arranged. More precisely, we can define permutation invariance in the following way:

Definition 4.2. A function $f : \mathbb{R}^{N\times d} \to \mathcal{Y}$ is invariant with respect to the permutation group Σ_N if $f(\mathbf{PX}) = f(\mathbf{X})$ for all $\mathbf{X} \in \mathbb{R}^{N\times d}$ and permutation matrices $\mathbf{P} \in \mathbb{R}^{N\times N}$. Similarly, if $\mathcal{Y} = \mathbb{R}^{N\times d'}$, f is said to be equivariant with respect to Σ_N if $f(\mathbf{PX}) = \mathbf{P}f(\mathbf{X})$.

Designing a permutation-invariant neural network puts severe restrictions on what types of layers can be used, but fortunately we can use our recipe of combining equivariant and invariant layers in order to achieve this. In general, we can describe the set of permutation invariant functions using a decomposition of two functions. Let $\psi : \mathbb{R}^d \to \mathbb{R}^M$ be a function that acts on individual set elements $\mathbf{x}_u$, $u \in \{1, ..., N\}$, i.e. the rows of $\mathbf{X}$, and let $\phi : \mathbb{R}^M \to \mathcal{Y}$. A permutation equivariant function can then be realized as:

$$f(\mathbf{X}) = \phi\left(\square_u \psi(\mathbf{x}_u)\right), \tag{27}$$

where $\square$ is any permutation-invariant aggregation operation, for example summation, in which case we say that this is a sum-decomposition of f via the latent space $\mathbb{R}^M$. This suggests that we could design an invariant neural network in this way by equipping ψ and ϕ with their own set of layers and learnable parameters, a method which is known as Deep Sets [29] and is illustrated in Figure 4. A similar method, using the max-decomposition as aggregation function, has also been proposed [18].

A natural question to ask is then if this design will satisfy universal approximation and, if so, are there any restrictions on the dimension of the latent space M? Although, there is no definitive answer under general conditions, some important results exist if we restrict ourselves to scalar sets ($d = 1$) and continuous functions with codomain $\mathcal{Y} = \mathbb{R}$.

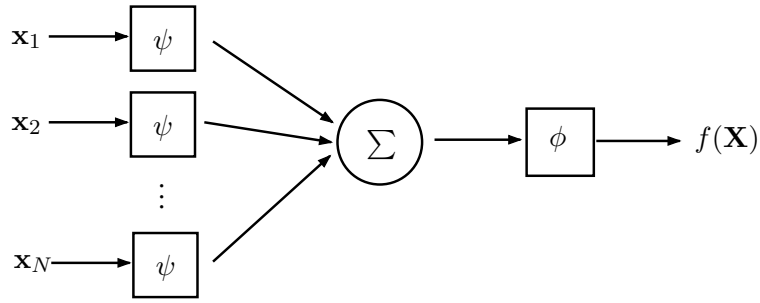

Figure 4: An illustration of the Deep Sets [29] architecture. Each set element is processed by the same neural network ψ and the outputs are aggregated using summation. The aggregated output is then fed into another neural network ϕ that predicts a global property of the set. Note that summation can be replaced by any other permutation-invariant aggregation.

Theorem 4.3. Wagstaff et al. (2019) [25]. Let $M > N$. Then there exist permutation invariant continuous functions $f : \mathbb{R}^M \to \mathbb{R}$ which are not sum-decomposable via $\mathbb{R}^N$.

The theorem gives us a necessary condition on the latent space in order to make sure that all functions can be represented, although clearly some functions exist that can be represented with a smaller latent space. In particular, the minimum size of the latent space depends on the number of elements in the set, such that larger input sets require a larger latent space. Furthermore, the following theorem proves that a latent space of size N is not only necessary, but also sufficient:

Theorem 4.4. Wagstaff et al. (2019) [25]. Let $f : \mathbb{R}^M \to \mathbb{R}$ be continuous. Then f is permutation-invariant if and only if it is continuously sum-decomposable via $\mathbb{R}^M$.

The reader is referred to [25] for proofs of theorems 4.3 and 4.4, which together have the following important implication: the Deep Sets architecture is guaranteed to have universal continuous function representation on sets of size N if and only if the dimension of the latent space is at least N. Although these theorems deal with universal representation, they have also been adapted for universal approximation with the same conclusions [26]. It should also be noted that this does not imply that the function approximation can always be practically implemented using a neural network, since the choice ψ and ϕ together with the optimization procedure might result in a network that in practice does not converge to the true function. Nevertheless, it gives a strong suggestion that the latent space of the network should grow with the number of elements in the input set.

4.2 Learning on Graphs

Learning on sets can be viewed as a special case of learning on graphs. Consider the graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, with vertices $\mathcal{V} = \{u\}_{u=1}^N$ and directed edges $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$. Each vertex

u of the graph can then be assigned certain features using row-vectors $\mathbf{x}_u$ and the edges model interaction between the vertices. A graph neural network (GNN) can then learn to extract features from the graph and predict properties of the vertices and edges, as well as the entire-graph. Applications of GNNs include modelling of social networks, where vertices are members of the network with their associated features and edges represent interactions between members, and drug discovery, where graphs are used to model interactions between atoms in a molecule [1].

In this context, learning on sets is a special case of learning on graphs, where the set of edges is empty, i.e. $\mathcal{E} = \emptyset$. However, if the set of edges is non-empty, we can use the adjacency matrix $\mathbf{A}$, which has elements of the form

$$a_{u,v} = \begin{cases} 1, & (i,j) \in \mathcal{E} \\ 0, & \text{otherwise.} \end{cases} \tag{28}$$

For sake of brevity, we only consider undirected graphs where $\mathbf{A} = \mathbf{A}^T$. Using this notation, a GNN can in general be described by a function that takes both the vertex features $\mathbf{X}$ and the adjacency matrix $\mathbf{A}$ as input. Similarly as for sets, we require permutation invariance for GNNs, since in general it is not possible to order the vertices of the graph in a well-defined manner. Since a permutation of the rows of $\mathbf{X}$ implies a permutation of both the rows and columns of $\mathbf{A}$, we can define the notion of permutation invariance for functions on graphs as follows:

Definition 4.5. Let $\mathcal{A}$ be the set of adjacency matrices of all the corresponding graphs of cardinality $N = |\mathcal{V}|$. A function $f : \mathbb{R}^{N\times d} \times \mathcal{A} \to \mathcal{Y}$ is invariant with respect to the permutation group Σ_N if $f(\mathbf{PX}, \mathbf{PAP}^T) = f(\mathbf{X}, \mathbf{A})$ for all $\mathbf{X} \in \mathbb{R}^{N\times d}$ and permutation matrices $\mathbf{P} \in \mathbb{R}^{N\times N}$. Similarly, if $\mathcal{Y} = \mathbb{R}^{N\times d'}$, f is said to be equivariant with respect to Σ_N if $f(\mathbf{PX}, \mathbf{PAP}^T) = \mathbf{P}f(\mathbf{X}, \mathbf{A})$.

GNNs are often realized using various forms of message-passing. This means that, in each layer l of the network, the features $h^l(x_u)$ of each vertex u are updated with messages from their neighbouring vertices $\mathcal{N}_u = \{v : (u,v) \in \mathcal{E}\}$. Letting $h^0(\mathbf{x}_u) = \mathbf{x}_u$, this can be written as

$$h^l(\mathbf{x}_u) = \phi\left(h^{l-1}(\mathbf{x}_u), \square_{v\in\mathcal{N}_u} \psi(h^{l-1}(\mathbf{x}_u), h^{l-1}(\mathbf{x}_v))\right), \quad l = 1, ..., L. \tag{29}$$

Note that we require permutation invariance with respect to the ordering of the vertices in $\mathcal{N}_u$. The message passing is then repeated for multiple layers and if the graph is connected,

i.e. there exists a path between every pair of vertices, then information can propagate between all vertices, given that there are sufficiently many layers in the network. If the goal is to predict a global feature of the graph, we can of course aggregate over all vertices in the graph in the last layer L as

$$f(\mathbf{X}) = \varphi\left(\square_{u} h^L(\mathbf{x}_u)\right). \tag{30}$$

Here we can make the observation that Deep Sets is a special form of GNN, because if $\mathcal{E} = \emptyset$ then each layer (29) simplifies to an update for each individual set element, without any interaction between different elements of the set.

4.3 Self-Attention and The Transformer

Self-attention is a special form of message-passing where the messages passed from v to u only depend on the features of v, but each message is weighted using a scalar attention weight

$$h^l(\mathbf{x}_u) = \phi\left(h^{l-1}(\mathbf{x}_u), \square_{v\in\mathcal{N}_u} \alpha(h^{l-1}(\mathbf{x}_u), h^{l-1}(\mathbf{x}_v))\psi(h^{l-1}(\mathbf{x}_v))\right). \tag{31}$$

The attention weight $\alpha(h^{l-1}(\mathbf{x}_u), h^{l-1}(\mathbf{x}_v))$ can be interpreted as a form of "soft" adjacency, indicating the strength of the connection between vertices u and v. Although there are many ways to implement self-attention, the most popular method is to calculate the attention weights using the scaled dot product of linear projections of the features. Let $\mathbf{q}_u = \mathbf{x}_u\mathbf{W}_q$ and $\mathbf{k}_v = \mathbf{x}_v\mathbf{W}_k$ denote the queries and keys, where $\mathbf{W}_q, \mathbf{W}_k \in \mathbb{R}^{d\times d_h}$ are learnable parameters of the model. The raw attention scores are calculated as

$$s_{uv} = \frac{\mathbf{q}_u\mathbf{k}_v^T}{\sqrt{d_h}}, \tag{32}$$

and the scaled dot-product attention weights are then given by

$$\alpha(\mathbf{x}_u, \mathbf{x}_v) = \frac{e^{s_{uv}}}{\sum_{v'\in\mathcal{N}_u} e^{s_{uv'}}}. \tag{33}$$

Note that due to the softmax normalization, we have that $\alpha(\mathbf{x}_u, \mathbf{x}_v) > 0$ and $\sum_{v \in \mathcal{N}_u} \alpha(\mathbf{x}_u, \mathbf{x}_v) = 1$. The message generating function in (31) is here simply defined as another linear projection $\psi(\mathbf{x}_v) = \mathbf{x}_v \mathbf{W}_v$ where $\mathbf{W}_v \in \mathbb{R}^{d \times d_h}$.

The scaled-dot product attention was originally proposed in [23] in order to improve natural language processing (NLP) models. In the context of NLP, self-attention is applied to sequences of words, which allows the model to exploit complex word-to-word interactions when processing the text. The connection between self-attention and GNNs here is subtle, but when self-attention is applied to the entire sequence, this can be regarded as modelling the sequence as a complete graph, i.e. $\mathcal{N}_u = \mathcal{V}$, $\forall u$. Using the complete graph allows us write the self-attention weights in matrix form as

$$\tilde{\mathbf{A}}(\mathbf{X}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_h}}\right), \tag{34}$$

where $\mathbf{Q} = \mathbf{X}\mathbf{W}_q$, $\mathbf{K} = \mathbf{X}\mathbf{W}_k$ and the softmax function is applied on each row individually. With this notation it follows that $\tilde{\mathbf{A}}_{uv} = \alpha(\mathbf{x}_u, \mathbf{x}_v)$, and we can define the self-attention (SA) operator as

$$\text{SA}(\mathbf{X}) = \tilde{\mathbf{A}}(\mathbf{X})\mathbf{V}, \tag{35}$$

where $\mathbf{V} = \mathbf{X}\mathbf{W}_v$, $\mathbf{W}_v \in \mathbb{R}^{d \times d_h}$. In order to make the operation even more expressive, it can be performed several times in parallel to for the multi-headed self-attention (MSA) operator

$$\text{MSA}(\mathbf{X}) = [\text{SA}_1(\mathbf{X}); \text{SA}_2(\mathbf{X}); ...; \text{SA}_h(\mathbf{X})]\mathbf{W}_P, \tag{36}$$

where ; denotes column-wise concatenation and $\mathbf{W}_P \in \mathbb{R}^{hd_h \times d}$. GNNs based on multi-headed self-attention are known as Transformers, and they have become the key component in NLP models, most notably the 175-billion parameter generative pre-trained Transformer (GPT) [2]. However, since the MSA operation is permutation invariant by design, it would not be able to distinguish between sequences of words where words have been re-ordered. Therefore, various forms of positional encodings that contain information about the relative position of the word in the sequence have been proposed. These encodings are then added to each word embedding before being processed by the Transformer.

So far, we have shown that the self-attention mechanism is one of many possible message-passing functions that can be derived from permutation invariance on sets or graphs, even though it was not originally proposed in this context. We shall now move on to discuss some problems where it is not obvious that self-attention is better than alternative methods, nor trivial to adapt to the particular setting.

4.4 Applications of Self-Attention

When designing deep networks, the family of architectures considered usually depends on the input domain $\mathcal{X}$. For example, the problem of image classification has the input domain of RGB images of a particular resolution, i.e. $\mathcal{X} = \mathbb{R}^{h \times w \times 3}$. Since the domain is a three-dimensional grid of pixels, we can use convolutions and pooling with local receptive fields in order to design networks that are either invariant or equivariant with respect to the location of objects in the picture. The representation of an image on a two-dimensional grid is intuitive, in the sense that it corresponds well with the way that we as humans perceive them with our eyes. However, this does not necessarily imply that this particular representation is optimal for neural network learning. For example, the image can just as well be represented on the one-dimensional grid $\mathbb{R}^{3hw}$ by stacking the pixel values in a vector. Furthermore, an image can also be represented as an unordered set $\{\mathbf{x}_u\}_{u=1}^{hw}$, where each element $\mathbf{x}_u = [r^{(u)}, g^{(u)}, b^{(u)}, p_x^{(u)}, p_y^{(u)}]$ stores information about the color intensity and coordinates of the pixels. From a human perspective, they appear to be counter intuitive representations, but from a deep learning perspective, it is not obvious if they are better or worse suited for learning.

More generally, we can describe images, and many other types of data, as graphs. If we extract square patches from an image with resolution (P, P), then we can represent each patch $\mathbf{x}_u$ on the domain $\mathbb{R}^{CP^2}$, yielding a total of $N = hw/P^2$ patches. We can then construct a graph $\mathcal{G} = (\{\mathbf{x}_u\}_{u=1}^{N}, \mathcal{E})$ by defining the set of edges $\mathcal{E}$. There are several ways to do this, and the edges of the graph will be redundant if we also use positional encodings for each patch. One possibility is to define edges between neighbouring patches in the image. Another possibility is to assign edges based on the semantic contents of the patches, i.e. let patches with similar objects be connected, but it is not clear how the semantic similarity would be defined. Finally, it is also possible to let the graph be complete and let all patches have edges between them, and then *learn* the strength of the connections, which is exactly what a Transformer does.

Recent work in computer vision [11, 22] suggests that the patch-based representation of images might be better suited for deep learning than a grid. By treating an image like a complete graph of small patches, and applying self-attention to infer the strength of the connections, Transformer-based networks can achieve equal or stronger performance in image classification tasks compared to convolutional networks. This is in many ways remarkable, since the Transformer-based network is not inherently translation invariant, and was not originally designed for solving problems in computer vision.

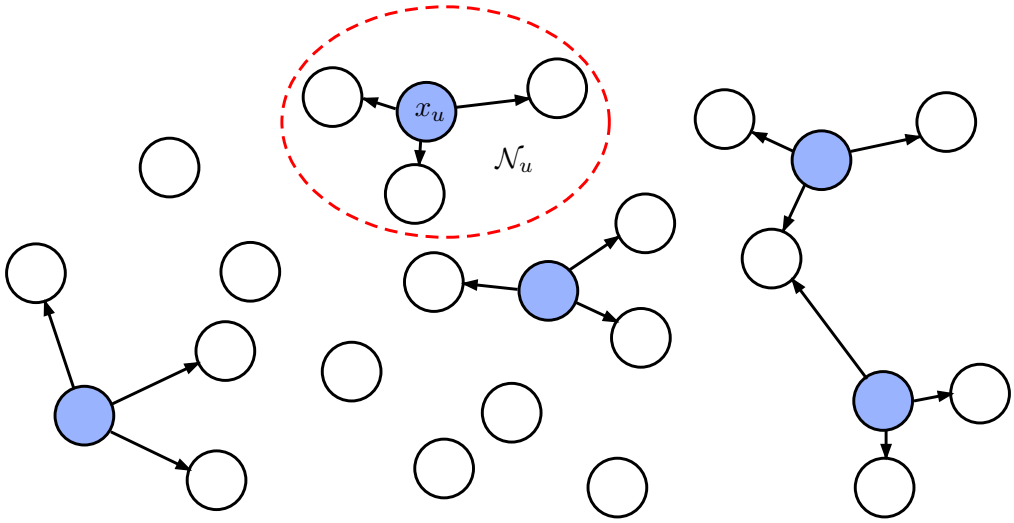

Figure 5: An example of a sparse k-NN graph created from a point cloud. Each anchor point (shown in blue), is connected with a directed edge from itself to its neighbours.

Point Cloud Feature Extraction and SLAM

An important application of GNNs is point cloud processing. Point clouds are frequently occurring in computer graphics, but also from measurements collected by 3D scanners, such as radar or LiDAR. Common applications include simultaneous localization and mapping (SLAM), which can be used for indoor navigation, and driver assistants in vehicles. In both of these scenarios, point clouds are used for creating maps of the environment. In order to create global maps, point clouds from different measurements need to be registered in a common frame of reference. This requires local point features that are resilient to outliers, since the measured data often contain noise and artifacts. The features can be based on purely geometrical properties, but with the advent of deep learning it is also possible to extract semantic features that describe the contents of the observations. Furthermore, such features can also be used for scene segmentation and object detection, which is used in self-driving cars for detecting e.g. lanes, signs and other vehicles.

A point cloud is an unordered set $\{\mathbf{x}_u\}_{u=1}^N$ of 3D Cartesian coordinates, $\mathbf{x}_u = [p_x^{(u)}, p_y^{(u)}, p_z^{(u)}]$. Deep networks for point cloud processing can be designed to extract global features that can be used for downstream tasks such as classification and retrieval, but also point-wise features that can be used for segmentation. Pioneering works in the field include PointNet [18] and Deep Sets [29], which used permutation invariant aggregation over the entire point cloud. It was also shown that the network satisfies universal approximation for a large enough dimension of the latent space, a result which is closely related to Theorems 4.3 and 4.4. Nevertheless, networks that treat the point cloud as an unordered set come with

some limitations, namely that they do not allow interactions of neighbouring points. This motivates the introduction of GNNs, which can capture local interaction more effectively.

In order to model the point cloud as a graph, we can use the k-nearest neighbours (k-NN) of x_u. Specifically, the k-NN graph of the point cloud is defined by edges between $\mathbf{x}_u$ and $\mathbf{x}_v$ if $\mathbf{x}_v$ is one of $\mathbf{x}_u$:s k-nearest neighbours in terms of Euclidean distance. We can then use GNNs with message-passing from all the neighbours of each point to the point itself. Other forms of local connectivity is also possible, for example by allowing edges from $\mathbf{x}_u$ to all points within a certain distance. These ideas were first explored in [19] and [27], where the latter also considered dynamic graphs that are updated in various stages of the network.

We may also choose to model point clouds using complete graphs with message passing between all points using self-attention, which was first proposed in [13]. However, the self-attention matrix $\tilde{\mathbf{A}}$ in (34) for N points will have size $N \times N$, which implies that the computational and memory requirements of the network will grow as $\mathcal{O}(N^2)$. For large inputs (point clouds often have thousands of points), using the complete graph will therefore not be computationally efficient.

Furthermore, it is not obvious that self-attention is useful when applied on individual point features, since a single point contains no semantic information. A natural extension is therefore to consider attention between patches of points, since a collection of neighbouring points can describe objects or parts of objects. One way to do this is to sample a sparse set of anchor points and create patches by aggregating over their neighbours, as shown in Figure 5. Applying attention between these patches would both decrease the computational complexity and capture more semantically meaningful connections between, compared to using the complete graph. These ideas are explored in more detail in Paper III , where they are also verified by extensive experiments on both real and synthetic data.

Audio Classification

Another important application of deep learning, which we have not dealt with so far, is audio processing, which encompasses a wide set of tasks related to feature extraction from audio waveforms. In particular, deep learning is well suited for tackling problems related to automatic speech recognition (ASR), where it is difficult to derive hand-crafted features that describe the contents of the speech signal. By leveraging large amounts of training data together with deep neural networks, ASR has now become a standard feature of our daily lives, for example in smart voice assistants.

An example of a smart assistant pipeline is shown in Figure 6. The first step in the process is wake-word detection, which is often done in an always on manner, that triggers the device to analyze the audio further. The second step is keyword spotting, where the goal is to

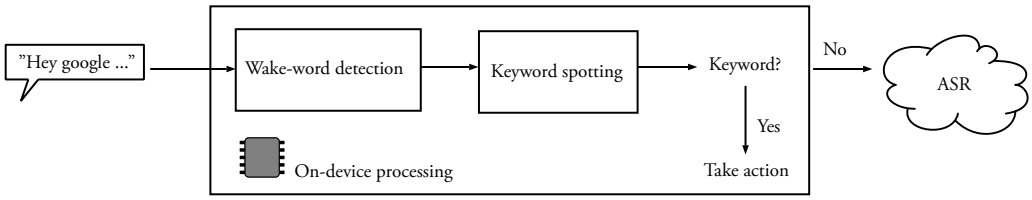

Figure 6: An example speech processing pipeline for smart assistants. Wake-word detection and keyword spotting are performed on-device. If no keyword is detected, the audio data is sent to a server where ASR is performed.

detect the presence of keywords from a small dictionary. Examples of keywords could be commands like "play" or "pause" in the context of listening to music. If no keyword is detected, then ASR is triggered, which is sometimes done on a remote server if on-device computational resources are limited. There are several advantages of performing each step locally on-device, the most obvious one being reduced latency. Data privacy can also be of concern and local audio processing reduces the amount of potentially sensitive data that needs to be transmitted to the server.

Here we restrict our analysis to keyword spotting, which can be regarded as a classification task where the goal is to classify the audio as one of the keywords in the dictionary or "silence" or "unknown". Current state-of-the-art keyword spotting methods do not operate on the raw audio input, but on a filtered and pre-processed audio representation as shown in Figure 7. The most widely used pre-processing technique is to extract speech features in the form of mel-frequency cepstrum coefficients (MFCCs). These are calculated by first using a sliding filter with a certain width and stride. The discrete Fourier transform is then calculated for each filtered output and the frequency amplitude are converted to mel-scale, which better captures differences in frequencies as perceived by humans compared to a Hz-scale. The final MFCCs are obtained by taking the discrete cosine transform of the log mel-scale spectrum.

Let $\mathbf{x}_n \in \mathbb{R}^d$ be the d output coefficients of the n:th time window. By stacking all time windows in a matrix as $\mathbf{X} = [\mathbf{x}_1, ..., \mathbf{x}_T]^T \in \mathbb{R}^{T\times d}$, we get something that resembles an image on a grid, with one grid axes for time and one for mel-scale frequency. Therefore, it is possible to borrow CNN architectures from computer vision and apply them to the speech features, which has been done extensively in order to provide state-of-the-art keyword spotting architectures [15].

However, given the surprising success of Transformers in computer vision, an interesting questions is whether this architecture also works well audio processing. Naturally, we can use a graph with a vertex for each time slot to describe the MFCC features. Unlike for the case of point clouds, the ordering of the time slots matter, so we need to add positional encoding to make the Transformer exploit this information. The reader is encouraged to read Paper 11 for a detailed description of this approach, experiments on human voice

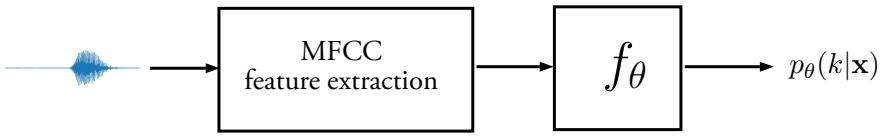

Figure 7: A keyword spotting pipeline. The raw audio waveform is first processed by extracting the MFCC spectrogram, which is then fed into a neural network that predicts a probability distribution over the different keywords.

recordings and comparisons of Transformers to CNNs.

5 Conclusions

We have now covered the necessary background theory for both regression ensembles and diversity, as well as GNNs and Transformers. These are the two main themes for the included publications in this thesis.

Paper I deals with applying the notion of label diversity to a wide range of problems in computer vision, where the data labels are both continuous and ordinal. By exploiting the possibility to combine different discretizations of continuous labels, and combinations of ordinal labels, we propose several strategies to induce label diversity in practice. Furthermore, we provide a method for implementing label diversity with minimal computational overhead that can be used in conjunction with standard neural network feature extractors. The method is based on using multiple prediction heads, one for each label representation. During the training phase, each head learns to classify the data into a specific set of labels, and during inference the predictions are combined in an ensemble-like fashion. From the ambiguity decomposition (15), we know that the prediction error of the ensemble average is guaranteed to be smaller than the average individual prediction error. By thorough experiments we also show that label diversity can reduce the prediction errors compared to standard methods like regression and RvC.

In Paper II and III we deal with the application of Transformer networks to different data domains. In Paper II , we propose the first Transformer-based keyword spotting model that achieves state-of-the-art results on common benchmarks. By ablation studies we highlight the benefit of using audio time slots as input patches to the model, and we show that the model learns to attend to the time slots that are most important for classification. In addition, we measure latency on a mobile phone and show that Transformers are competitive in this regard as well. However, there remains work to be done with regards to model compression in order to make Transformers a viable alternative in low-footprint edge use cases. In future work, we wish to study the effects of common compression techniques, such as pruning and quantization, on Transformer based keyword spotting models.

In Paper III , we instead apply Transformers to point cloud processing, where the main

problem is to reduce the quadratic complexity of the self-attention operator. We do this by using a hierarchical Transformer that applies self-attention locally and globally in a two-stage process. Experiments show that this not only reduces computational footprint, but also makes the features of the Transformer better suited for downstream tasks such as classification and segmentation. Finally, we also show that the proposed method can be used to improve feature matching between point clouds, which is commonly used in SLAM and other applications.

References

[1] Michael M. Bronstein, Joan Bruna, Taco Cohen, and Petar Veličković. Geometric deep learning: Grids, groups, graphs, geodesics, and gauges, 2021.

[2] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[3] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[4] Raul Diaz and Amit Marathe. Soft labels for ordinal regression. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4738–4747, 2019.

[5] Mario AT Figueiredo. Adaptive sparseness using jeffreys prior. In *NIPS*, pages 697–704, 2001.

[6] Eibe Frank and Mark Hall. A simple approach to ordinal classification. In *European conference on machine learning*, pages 145–156. Springer, 2001.

[7] Yoav Freund and Robert E Schapire. Experiments with a new boosting algorithm. Citeseer, 1996.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[9] Ian Goodfellow, Oriol Vinyals, and Andrew Saxe. Qualitatively characterizing neural network optimization problems. In *International Conference on Learning Representations*, 2015.

[10] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[11] Alexander Kolesnikov, Alexey Dosovitskiy, Dirk Weissenborn, Georg Heigold, Jakob Uszkoreit, Lucas Beyer, Matthias Minderer, Mostafa Dehghani, Neil Houlsby, Sylvain Gelly, Thomas Unterthiner, and Xiaohua Zhai. An image is worth 16x16 words: Transformers for image recognition at scale. 2021.

[12] Anders Krogh, Jesper Vedelsby, et al. Neural network ensembles, cross validation, and active learning. *Advances in neural information processing systems*, 7:231–238, 1995.

[13] Juho Lee, Yoonho Lee, Jungtaek Kim, Adam Kosiorek, Seungjin Choi, and Yee Whye Teh. Set transformer: A framework for attention-based permutation-invariant neural networks. In *International Conference on Machine Learning*, pages 3744–3753. PMLR, 2019.

[14] Ling Li and Hsuan-Tien Lin. Ordinal regression by extended binary classification. In *Advances in neural information processing systems*, pages 865–872, 2007.

[15] Iván López-Espejo, Zheng-Hua Tan, John Hansen, and Jesper Jensen. Deep spoken keyword spotting: An overview, 2021.

[16] Preetum Nakkiran, Gal Kaplun, Yamini Bansal, Tristan Yang, Boaz Barak, and Ilya Sutskever. Deep double descent: Where bigger models and more data hurt. In *International Conference on Learning Representations*, 2020.

[17] Michael P Perrone and Leon N Cooper. When networks disagree: Ensemble methods for hybrid neural networks. Technical report, BROWN UNIV PROVIDENCE RI INST FOR BRAIN AND NEURAL SYSTEMS, 1992.

[18] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 652–660, 2017.

[19] Charles Ruizhongtai Qi, Li Yi, Hao Su, and Leonidas J Guibas. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. *Advances in Neural Information Processing Systems*, 30, 2017.

[20] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[21] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[22] Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Hervé Jégou. Training data-efficient image transformers & distillation through attention. In *International Conference on Machine Learning*, pages 10347–10357. PMLR, 2021.

[23] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 6000–6010, 2017.

[24] Athanasios Voulodimos, Nikolaos Doulamis, Anastasios Doulamis, and Eftychios Protopapadakis. Deep learning for computer vision: A brief review. *Computational intelligence and neuroscience*, 2018, 2018.

[25] Edward Wagstaff, Fabian Fuchs, Martin Engelcke, Ingmar Posner, and Michael A Osborne. On the limitations of representing functions on sets. In *International Conference on Machine Learning*, pages 6487–6494. PMLR, 2019.

[26] Edward Wagstaff, Fabian B. Fuchs, Martin Engelcke, Michael A. Osborne, and Ingmar Posner. Universal approximation of functions on sets, 2021.

[27] Yue Wang, Yongbin Sun, Ziwei Liu, Sanjay E Sarma, Michael M Bronstein, and Justin M Solomon. Dynamic graph cnn for learning on point clouds. *Acm Transactions On Graphics (tog)*, 38(5):1–12, 2019.

[28] Jack H Winters, Jack Salz, and Richard D Gitlin. The impact of antenna diversity on the capacity of wireless communication systems. *IEEE transactions on Communications*, 42(234):1740–1751, 1994.

[29] Manzil Zaheer, Satwik Kottur, Siamak Ravanbhakhsh, Barnabás Póczos, Ruslan Salakhutdinov, and Alexander J Smola. Deep sets. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 3394–3404, 2017.

[30] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning (still) requires rethinking generalization. *Communications of the ACM*, 64(3):107–115, 2021.