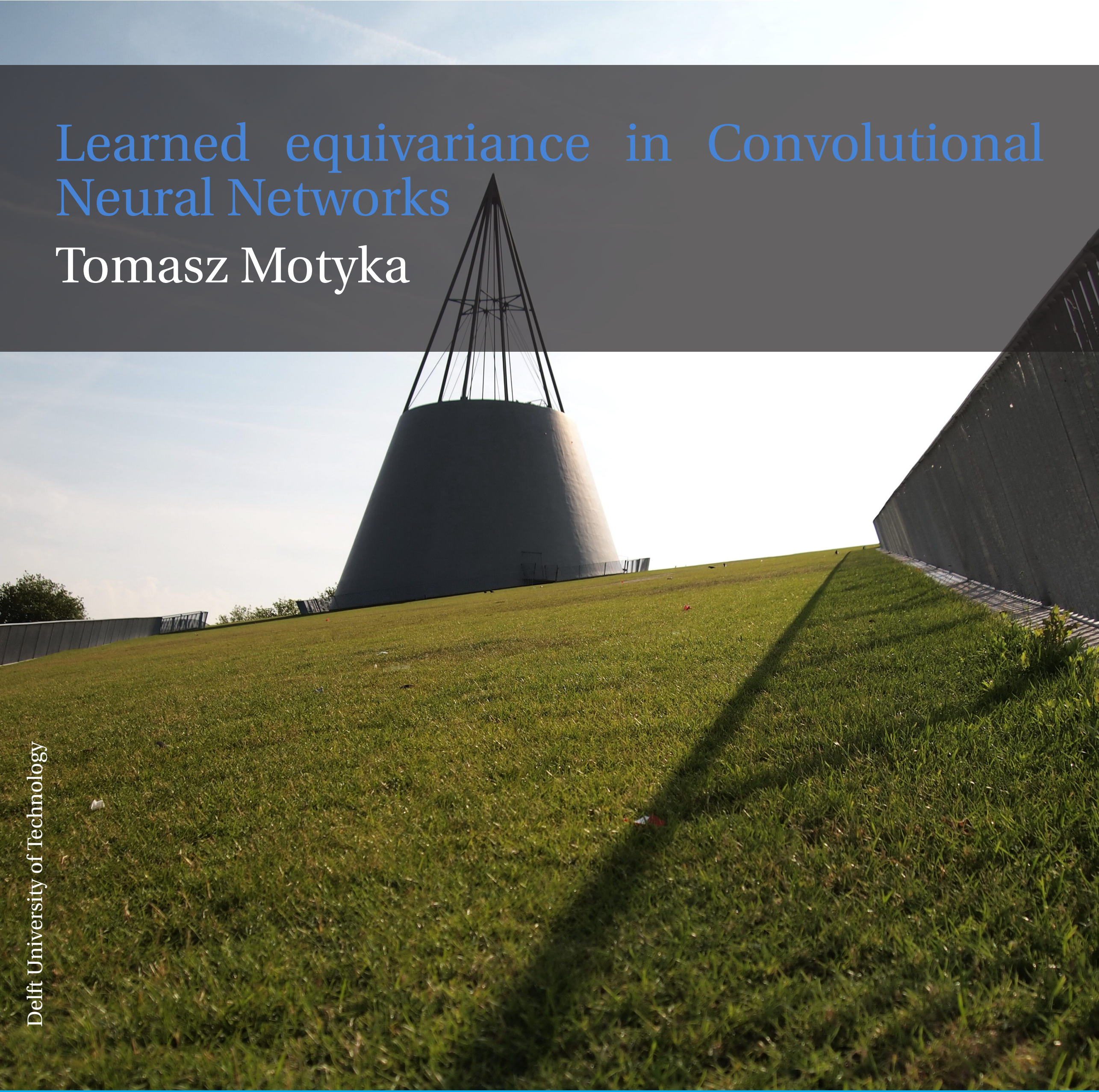
Learned equivariance in Convolutional Neural Networks

Tomasz Motyka



Delft University of Technology

TU Delft

Learned equivariance in Convolutional Neural Networks

by

Tomasz Motyka

to obtain the degree of Master of Science
at the Delft University of Technology
to be defended publicly on Wednesday January 12, 2022 at 9:00 AM

Student number:	5146844	
Project duration:	November 15, 2020 - January 12, 2022	
Thesis committee:	dr. J. C. van Gemert	TU Delft, supervisor
	Dr. M.M. de Weerdt	TU Delft, External thesis committee member

Preface

The current report "Learned Equivariance in Convolutional Neural Networks” presents work done for my master’s graduation project. The research was conducted within Computer Vision Lab at TU Delft, under the supervision of Dr. J.C. van Gemert as the main supervisor and Robert-Jan Bruintjes as the daily supervisor.

I would like to thank Jan for the support in brainstorming research ideas, pointing directions and showing how to ask the right questions. Additionally, I would like to thank Robert-Jan for his tremendous support in our weekly meetings and motivation in times of doubt. It was priceless, particularly in the time of online work, to have someone understanding your subject with whom I could always have a valuable discussion when necessary. I am also thankful for giving me access to the TU Delft computing cluster, which was extremely useful for conducting the experiments.

I would like to thank Dr. M.M. de Weerdt, for his interest in my thesis and evaluation of my work.

I would like to thank my parents for their emotional and financial support in the last two years. Time spent in Delft wouldn’t be possible without them. I would also like to thank my fiance for her immense support, patience and motivation on a daily basis throughout my time in Delft. Finally, I would also like to thank my friends for valuable discussions and mutual support.

Tomasz Motyka
Delft, January 2022

Contents

1
Research Paper

Learned equivariance in Convolutional Neural Networks

Tomasz Motyka
Computer Vision Lab
Delft University of Technology
t.t.motyka@student.tudelft.nl

Abstract

Aside from developing methods to embed the equivariant priors into the architectures, one can also study how the networks learn equivariant properties. In this work, we conduct a study on the influence of different factors on learned equivariance. We propose a method to quantify equivariance and argue why using the correlation to compare intermediate representations may be a better choice as opposed to other commonly used metrics. We show that imposing equivariance or invariance into the objective function does not influence learning more equivariant features in the early parts of the network. We also study how different data augmentations influence translation equivariance. Furthermore, we show that models with lower capacity learn more translation equivariant features. Lastly, we quantify translation and rotation equivariance in different state-of-the-art image classification models and analyse the correlation between the amount of equivariance and accuracy.

1. Introduction

Neural network $f(x)$ is invariant to the transformation T if applying T to the input x does not change the network output. In other words, $f(T(x)) = f(x)$. Alternatively, a network is equivariant to transformation T if its output changes predictably when x is transformed by T. Formally, we can define it as $f(T(x)) = T(f(x))$. Invariance in the predictions of machine learning models is important as it ensures predictable response with respect to different transformations of the input and helps to perform well on unseen and potentially out-of-distribution samples. The equivariance property, on the other hand, is more powerful as besides giving us a predictable representation for different input transformations it also contains the information about the transformation itself (Figure 1).

Deep Neural Networks (DNNs) have benefited greatly from the incorporation of different equivariant priors. Such prior knowledge can provide certain properties [4, 8] or it can greatly improve the data efficiency [4, 8, 13]. The most notable example of such a prior is the introduction of the convolution operation into DNNs. Convolutional Neural Networks (CNNs) [4], thanks to their (approximate [18, 20, 44]) translation equivariance, have proven to be very powerful models for the data types such as images or audio [4, 8]. Other works on equivariant architectures have gone even further to ensure equivariance to other affine transformations [8, 12, 28, 36]. However, explicit equivariant priors are not the only source of equivariance. Namely, CNNs can learn these properties themselves if certain symmetries are present in the data [5, 19, 24, 26, 27]. Studying learned equivariance can tell us which factors of the experiment e.g. model architecture, data augmentations or objective function, do affect equivariance or it can give insights on how to design equivariant priors. However, most of the works that study learned equivariance focus on investigating invariant properties at the final layer, meaning that they measure whether different input transformations do influence the prediction [5, 19, 25]. There are few works that study equivariant properties of intermediate representations but they usually focus on showing how translation equivariance, which should be given by the model, is violated. What's missing in the literature is more comprehensive study on how various factors affect the equivariance of intermediate representations. In this work we analyze how to quantify equivariance of the internal representations. We study how symmetries that are in the data influence learning equivariance. We investigate

$$\begin{tikzcd}
x \arrow[r, "f"] \arrow[d, "T"'] & f(x) \arrow[d, "T"] \\
T(x) \arrow[r, "f"] & f(T(x)) = T(f(x))
\end{tikzcd}
\qquad
\begin{tikzcd}
x \arrow[rd, "f"] \arrow[dd, "T"'] & \\
& f(x) \\
T(x) \arrow[ru, "f"'] &
\end{tikzcd}$$

Figure 1: Difference between invariance and equivariance. On the left function f has the equivariance property w.r.t. to transformation T. On the left the function f has the invariance property w.r.t. to transformation T.

how translation equivariance is affected by different data augmentations and by model capacity. Additionally, we quantify equivariance in few large scale state-of-the art seminal models and correlate the accuracy with equivariance.

Our contributions are:

- We show why the commonly used cosine similarity [44] may not be the best choice to compare feature maps and argue for the use of correlation.
- We show that encoding equivariance directly into the objective function does not induce learning more equivariant features in the early and middle layers of the network.
- We show that random crops and cutmix data augmentations increase translation equivariance while some other augmentations do not have discernible effect.
- We show that smaller models learn more equivariance.
- We give a summary of the amount of equivariance learned by different large scale seminal models and show that out of all the layers, it is mostly the equivariance in the final layer that correlates with accuracy.

2. Background & Related Work

2.1. Embedding Equivariance

A lot of works on equivariance focus on designing architectures that ensure the equivariance property over different transformations. Most notable is the work of [8, 12] where the authors give the theory on how to embed equivariance over any discrete symmetry group (e.g. rotation) with Group Equivariant Convolution. This work has a number of follow-ups where the Group Equivariant Convolutions is extended to other discrete or continuous groups [3, 2, 7, 36] or applied to self-attention models [31, 32, 33]. Another interesting branch of equivariant models are steerable filter networks [6, 14, 28, 41] There are also methods that, unlike the previously mentioned works, do not embed certain equivariant priors upfront but instead develop architectures that aim to automatically learn needed symmetries from the data [34, 45]. Lastly, the CNN can learn equivariant intermediate representations by itself, without explicit equivariant prior, which was hinted by [24, 26, 27]. In our work we focus on the analysis of the equivariant properties that are being learned by the network.

2.2. Measuring equivariance

Another branch of research on equivariance is the analysis of the equivariant or invariant properties of learned network representations. In [16], the authors measure the invariance of Deep Belief Networks by comparing magnitudes of activations. [19] studies the source of translation invariance of CNNs using Euclidean distance to compare final layer embeddings and find that data augmentation has a bigger effect on translation invariance than the network architecture. [25] replaces Euclidean distance with cosine similarity as a similarity measure and shows the impact of different kernel and padding size on translation invariance at the final layer. [5] explores the sources of invariance in models trained on ImageNet [10], and how these invariances relate to the factors of variation of ImageNet [10]. They compared the impact of different transformations (e.g. translation, scaling, rotations), architectural bias and the data itself. In [1] the authors showed that learned invariances do not transfer across classes. All the aforementioned works focus only on the invariance in the final layer of the network, whereas we focus on the intermediate representations as well.

There are very few works that study equivariant properties of intermediate representations and they mostly focus on showing how translation equivariance in CNN is violated. [44] measured the translation equivariance by computing cosine similarity between feature maps to show how pooling operation violates translation equivariance property. Other works like [18, 20], also study how translation equivariance property in CNNs is violated, although they measure it primarily by looking at accuracy achieved in some specific settings. [35] measure the invariance of intermediate representations using normalized cosine similarity to study the effect of pooling on deformation stability. In [26, 27] the authors performed a huge qualitative study by visualizing what kind of features the early layers of InceptionV1 [37] learn to detect. Alongside visualizing features, they also searched for certain patterns of relationship between features, which they called *circuits*, and showed that CNN learns equivariant representations. Although they showed the existence of different *circuits*, these were examples among thousands of features, which does not necessarily represent the behaviour of the whole network in a quantitative way.

In our works we take a quantitative approach to what [26, 27] do. We derive a measure for quantifying equivariance and show why cosine similarity used in e.g. [35, 44] may not be the preferred measure. We complement most of the works that study the invariance in the final layers [5, 19, 25] by studying the effect of different factors on the equivariance of intermediate representations.

3. Analysis

3.1. Embedded vs Learned equivariance

First, we want to highlight the difference between learned and embedded equivariance. Discrete convolution operation is known to be translation equivariant:

$$[[\Delta(I)] \star \psi](x) = [\Delta[I \star \psi]](x), \tag{1}$$

meaning that applying translation Δ to the image I and convolving it with the filter ψ has the same effect as convolving I with ψ and applying Δ to the output. Since the translation equivariance property comes from translating the filter ψ over the image, it will hold regardless of the content of the learned filter ψ. We also point out that, in practice, CNNs are not perfectly translation equivariant. [44] shows how pooling over a discrete grid can deteriorate the translation equivariance. [20] shows that CNN can learn filters that exploit border effects which results in the lack of equivariance. Therefore, in our work on the learned equivariance, we also study translation equivariance since the network can learn filters that strengthen or weaken the aforementioned effects.

Convolution does not assure equivariance to affine transformations other than translations. In those cases, the learned weights determine the equivariance property. Although in measuring the equivariance we look at the activations, also called feature maps, rather than the learned weights, we want to first analyze what properties the weights need to have for the convolution to be equivariant. It gives us a better interpretation of the results and intuition about how to measure equivariance.

Why not study the weights directly? If the goal is to detect equivariant or invariant features learned by the CNNs, one may want to look directly at the learned weights. There are, however, potential issues with this approach. The main one is that the activations at a certain layer are not only dependent on the weights at that particular layer but are determined by a complex superposition of all the weights from the previous layers, which can be difficult to capture in the analysis. By looking at the feature maps instead of the filters we omit this issue and still can reason about the feature that is being detected.

3.2. Learned equivariance in a single convolution

Here, we analyze what properties the weights have to follow for the single convolution operation to achieve equivariance to other affine transformations like rotations or mirroring. In [8] the authors derive the following equation:

$$[[T(I)] \star \psi](x) = [T[I \star T^{-1}(\psi)]](x), \tag{2}$$

which denotes the conditions under which the convolution is equivariant to transformation T. Comparing this with Eq. 1, one can see that the only difference is that on the right hand side we have $T^{-1}(\psi)$ instead of ψ. Having that in mind, we now identify 2 ways in which the CNN can achieve equivariance:

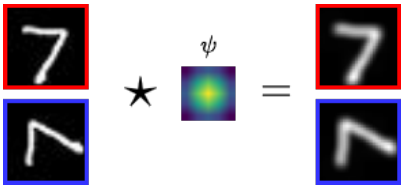


Figure 2: Convolution with rotationally invariant kernel, $\psi = T^{-1}(\psi)$ (e.g. 2D isotropic Gaussian) is rotation equivariant. Convolving the image with filter ψ (red border) gives a feature map with the same content as rotating the image and then convolving it with the filter ψW (blue border).

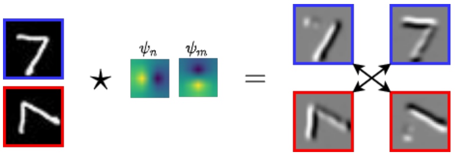


Figure 3: Convolving image with two filters following the property $\psi_n = T^{-1}(\psi_m)$ (blue border) results in the stack of feature maps with the same content as the stack of feature maps resulting from convolving the rotated image with those filters (red border). The difference is that the feature maps within the stack are ordered differently.

1. When the single filter ψ follows property $\psi = T^{-1}(\psi)$, which means it is invariant to transformation T^{-1}, the convolution becomes equivariant. We call this *channel equivariance*. In Figure. 2 we show an example where T is a 90° rotation.

2. In a single layer of a CNN, we have a set of multiple filters Ψ and so it also might be the case that there are filters $\psi_n, \psi_m \in \Psi$ that follow the property $\psi_n = T^{-1}(\psi_m)$. In such case the stack of resulting feature maps could be called equivariant whereas individual feature maps are not. We call this *layer equivariance*. In Figure. 3 we show an example where T is a 90° rotation.

3.3. Capturing equivariance

Measuring equivariance of intermediate representation requires comparing intermediate feature maps. In this section, we analyze how to compare feature maps in order to capture equivariance. Starting from the definition of the equivariance, we can simply compute the exact equality between the feature maps:

$$T(f(x)) = f(T(x)) \tag{3}$$

However, such formulation gives us a binary decision problem, where we either consider something equivariant or not. In the case of learned equivariance, it can rarely be the case that we will see exact equivariance with pixel-wise precision. That would require most of the filters to be either perfectly symmetric or different filters to be perfect rotated copies of one another. With random initialization of weights and non-uniform gradients, this is very improbable. What we would prefer instead is a continuous measure that would give us a score reflecting the approximate amount of equivariance, i.e.:

$$\text{Equivariance}(T(f(x)), f(T(x))) \in [0, 1] \tag{4}$$

A measure quite commonly used in the literature [5, 25, 44], is the cosine similarity which is based on the dot-product. The value of cosine similarity ranges between -1 and 1 where -1 stands for inverse similarity, 0 stands for no similarity/orthogonality, and 1 stands for perfect similarity. In our case, we are only interested in positive similarity since we want the feature maps to have the same content in case of equivariance and so we apply $ReLU()$ function on top of the score.

3.4. Method

So far, we have analyzed the difference between embedded and learned equivariance and showed how a single convolutional layer can achieve equivariance. Here, we describe our method for measuring equivariance in multi-layer CNNs.

We consider a CNN composed of a set of L convolutional blocks where each block consists of the convolution operation followed by non-linearity and often a batch normalization. Output of each convolutional block $f_l(x), \forall\, l \in L$ has K^l channels. We compute the equivariance of intermediate representations $f_l(x)$ using the output of such blocks since we consider a whole block as a feature detector and that is what the layer $l+1$ learns upon. Doing otherwise may lead to considering information that is irrelevant to the next layer since it is removed by non-linearity. It is also the case that different models may have different orderings of the operations within the block e.g. batch-norm happens before or after non-linearity. In principle, we measure:

- Channel equivariance, where for each channel index $k \in K^l$ at layer $l \in L$ we compute the similarity score between the feature maps: $\text{Equivariance}(T(f_l^k(x), f_l^{k'}(T(x)))$ (Figure 4a) and average the scores over all channels in a layer. We also point out that after the global pooling operation, which is usually present in the neural networks, the transformation T becomes the identity transformation since the feature maps have size 1×1. That means that after the global pooling, the channel equivariance measures invariance.
- Layer equivariance of a network f where for each channel index $k \in K^l$ at layer $l \in L$, we compute: $\max_{k' \in K^l} \text{Equivariance}(T(f_l^k(x), f_l^{k'}(T(x)))$ (Figure 4b) and average the scores over all channels in a layer.

4. Experiments

4.1. Method verification

4.1.1 Choosing a proper similarity measure

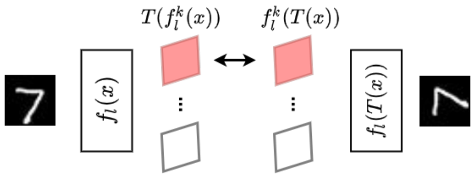
Cosine similarity has been a common similarity measure used to compute invariance or equivariance of the networks representations [5, 25, 44]. Here, we show why it may not be the proper choice.

Activations at different layers of the CNN may have different means. Cosine similarity is sensitive to the mean values of its input vectors. This behaviour is depicted in Figure 5a. Pearson correlation [15], also called centered cosine similarity, is a measure that alleviates this issue as it captures the similarity of the patterns between two input vectors. It is also the basis of many methods for comparing network representations [21, 23, 30]. For the same reason as in the case of cosine similarity, we apply the $ReLU()$ function on top of the score.

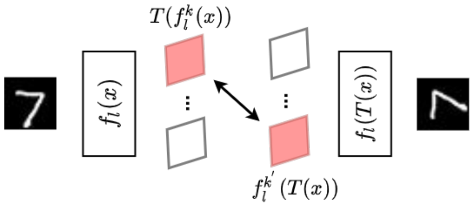
We visualize the difference between using cosine similarity and correlation for capturing equivariance in the following example. We train ResNet-44 [17] model on CIFAR-10 [22] dataset and compute channel equivariance w.r.t to 90° rotation after each residual block. In Figure 5 we show the qualitative comparison between the scores, computed using cosine similarity and correlation, and the mean of the activations. Additionally, we compute a correlation between the magnitude of the activations and equivariance scores computed with cosine similarity (**0.63**) and correlation (**0.11**). Scores computed using cosine similarity correlate visibly with the mean of the activation while, for the scores computed using correlation, this effect is less prevalent. In our experiments, we use correlation as a measure to quantify equivariance since it enables us to reliably compare the scores between layers.

4.1.2 Does the method capture equivariance

To verify, that our method captures equivariance, we apply it to the artificially created scenarios. We create two 3-layer CNN with hand-crafted weights such that we expect perfect channel and layer equivariance. In the first scenario, we set all the weights to be rotationally symmetric, using 2D isotropic Gaussian, and measure the channel equivariance after each layer (Fig. 6a). In the second scenario, we cut different corners of the filters from the previous setting such that all the filters are rotations of one another (Fig. 6b). This time we measure layer equivariance. In both cases, our method gives perfect scores confirming that it captures equivariance.

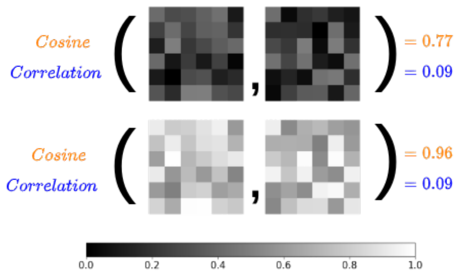


(a) Channel equivariance

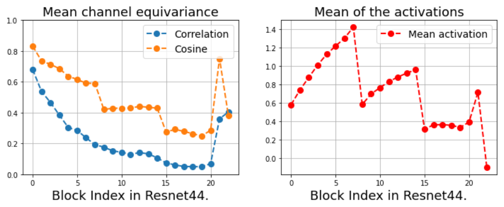


(b) Layer equivariance

Figure 4: Channel equivariance is measured between the corresponding channels while in case of layer equivariance we take the maximum from the scores computed between channel k from one representation and all the channels from the other representations .

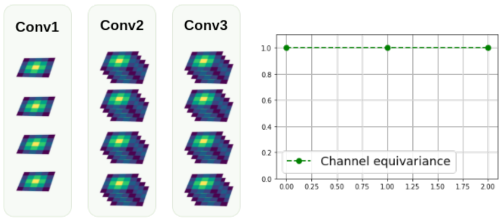


(a) Comparison of different similarity measures on random feature maps. Feature maps on the bottom are shifted by 0.5 with respect to the top ones. Cosine similarity is sensitive to such shifts while correlation is invariant.

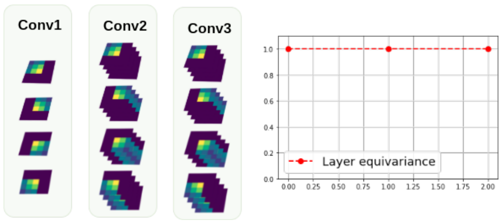


(b) Comparison between magnitude of the activations (red) and equivariance computed using correlation (blue) and cosine similarity (orange). We compute correlation between the magnitude of the activations and equivariance scores computed with correlation (**0.11**) and cosine similarity (**0.63**).

Figure 5: Analysis of the influence of magnitude of weights on different similarity measures.



(a) Rotationally symmetric filters result in perfect channel equivariance.



(b) When each layer consists of the same rotated filters, out methods gives perfect layer equivariance.

Figure 6: We create 3 layer CNN with handcrafted weights, such that it has perfect equivariance, to test if our method gives the highest scores.

4.2. Influence of hyperparameters on learning equivariance

4.2.1 Imposing symmetries in the data.

In this experiment we study how the equivariance changes if we impose symmetries in the data and therefore into the objective function. We also study whether there is a difference if the objective is invariant or equivariant with respect to introduced symmetries. We train a 7-layer CNN with 7 layers of 3×3 convolutions, 20 channels in each layer, ReLU activation functions, batch normalization, and max-pooling after layer 2, taken from [8], on 3 different datasets. The first dataset is *MNIST6*, which is the regular MNIST[11] without $\{0, 1, 6, 8\}$ classes, to get rid of rotational symmetries that these classes have. For example, digit 8 is very similar to its $180°$ rotation, so, by default, we have some rotation invariance. Second is the *MNIST6-Rot-Inv* where every digit in *MNIST6* is randomly rotated by $r \in \{0, 90, 180, 270\}$ upfront. This dataset imposes invariance into the objective function as, for every transformation, the output has to be the same. The last dataset, *MNIST6-Rot-Eq*, is created in the same way as *MNIST6-Rot-Inv* but now we are also predicting the rotation alongside the digit. This dataset imposes equivariance into the objective function. We compute channel and layer equivariance using 2000 images from the validation set. We average the score over $90°, 180°, 270°$ rotations. Each experiment is repeated three times using different random seed. The training setting used for this experiment is in Table 1.

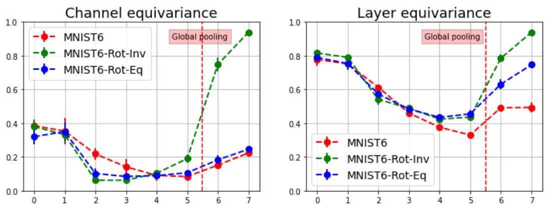


Figure 7: Channel (left) and layer (right) rotation equivariance for different objective functions. Equivariance or invariance in the task does not induce learning more equivariant features up until the late part of the network. Also there is no visible difference, up until the late part of the network, in the learned equivariance between equivariant and invariant task.

In Figure 7 we show the average channel and layer rotation equivariance. Firstly, we can observe that the amount of both channel and layer equivariance is decreasing with the depth, up to the final part after global pooling regardless of the task. For the tasks where the equivariance or invariance is imposed in the objective, we can see a spike in the final part, which may suggest that global pooling plays a significant role in achieving invariance or equivariance. Secondly, we do not see any significant differences between *MNIST6*, *MNIST6-Rot-Inv* and *MNIST6-Rot-Eq*, up to a later stage, which may hint that, in practice, early convolutional layers learn features with some amount of equivariance regardless of the invariance of the objective function. Finally, we observe that layer equivariance is much larger then the channel equivariance in the early and middle layers, which shows that CNNs do learn more rotated versions of the same feature in different channel rather then learning invariant, symmetrical features in a single channel.

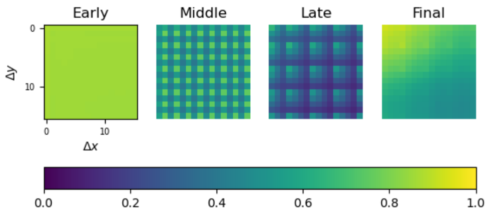


Figure 8: Translation equivariance for different parts of the Resnet44 trained on Cifar10 with no data augmentations. Feature maps in the *Middle* part of the networks are 2 times smaller and feature maps in the *Late* part of the networks are 4 times smaller. Such discrete reduction in size results in periodic violation of translation equivariance.

4.2.2 Data augmentation for translation equivariance

By duplicating input samples under some transformation, data augmentation induces invariance in the prediction. The question is what is the source of invariance. Does data augmentation result in more equivariant intermediate feature or the invariance is learned in some other way? For example, we know that random crops, which is essentially a translation transformation, do increase the model performance and translation invariance at the last layer [19], but do they increase the equivariance of learned features or just enable the CNN to exploit the border effects to a larger extent as shown in [20]? In this experiment we study how translation equivariance is affected by different data augmentations.

We train ResNet-44[17], adapted for CIFAR-10[22], on the CIFAR-10 datatset using one of the following augmentations: random crops, horizontal flips, CutMix [42], RandAugment [9]. In a single experiment we compute channel and layer equivariance using 2000 images from the validation set and average the score over diagonal shifts from 1 to 16 pixels. Each experiment is repeated 3 times using the network trained with different random seed. In principle we expect the layer and channel equivariance to be the same since translation equivariance should be provided by the convolution and so the maximum equivariance score should happen between the corresponding channels. However, we include layer equivariance in our experiments since there are works showing that the information about location can be encoded in different channels [18, 20]. The hyper parameters used for training are in the Table 2.

In Figure 8, we point out the effect of the reduction of the spatial dimension on the translation equivariance. If, for example, the intermediate representation is half the height and width of the input image and we shifted the input image by 5 pixels, then when measuring equivariance, we should shift the intermediate representation by 2.5 pixels to account for the reduction. Such operation does not exist on a 2D discrete grid. Thus, when translating the intermediate feature map, we resort to rounding down the shift size. None of the data augmentation methods solve this problem but rather shift the values up.

In Figure 9 we show the average translation channel and layer equivariance scores for different data augmentations. Random crops and RandAugment increase the equivariance of learned features in the middle, late and final parts while the other methods do not have any significant effect with cut mix even having less equivariance in the middle part. Furthermore, we complement the finding of [19] by showing that random crops increase not only translation invariance but also translation equivariance in the intermediate layers. Also, we do not see any difference between channel and layer equivariance for any data augmentation, which means that, in all cases, the network does not explicitly learn to represent different translations in different channels.

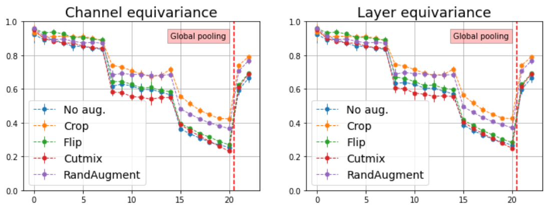


Figure 9: Channel (left) and layer (right) translation equivariance on CIFAR-10 for different data augmentation strategies. Random crops and RandAugment increase channel equivariance the most, while other strategies have no discernible improvements.

4.2.3 Model capacity for translation equivariance

Property of equivariance to transformation T gives us an efficient representation since we have a consistent representation when T is applied to any input. In this experiment, we study whether model capacity influences learning equivariant representations. We hypothesize that a smaller model in principle needs more efficient representation and hence may learn more equivariant features. To test this hypothesis we employ the family of WideResNet [43] models, where we scale the number of channels in each layer. We train 4 WideResnet40 models with 4 scale factors $s \in \{1, 2, 4, 8\}$ on Cifar10 dataset and measure channel and layer translation equivariance. The hyperparameters used for training are the same as in the data augmentation experiment of Sec 5.2 (Table 2).

In Figure 10 we show the average layer and channel translation equivariance for different scale factors. We observe that the amount of translation equivariance is lower for the wider models even though the amount of invariance in the final part is the same. This confirms the hypothesis that translation equivariance in the intermediate representations, which should be given by the model, improves when the model capacity is lower.

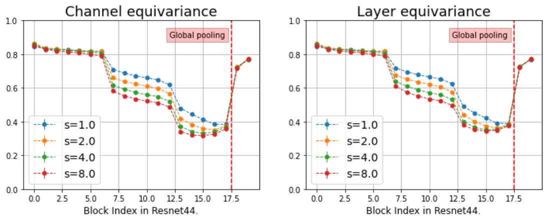


Figure 10: Channel (left) and layer (right) translation equivariance on CIFAR-10 for different model widths. Smaller models learn more equivariance although the amount of invariance in the end is similar.

4.3. Equivariance in large scale seminal models

So far, we've done the experiments on a small scale setting. In this experiment we want to study the amount of equivariance in the state of the art models from the recent years, trained on the ImageNet [10] dataset. In our experiment we are interested to what extent large scale models learn equivariance in the presence (translation) and absence (rotation) of equivariant prior and whether there are any substantial differences between the models. We will also look if the higher equivariance in the early layers induces higher invariance in the final layer. Finally, for rotation equivariance we want to see the difference between the layer and channel equivariance. This can tell us if the network learns to represent rotations in the same or different channels. We measure both translation and rotation equivariance on 2000 images from the validation set. We do not train the models ourselves but instead use available checkpoints from `torchvison` [29] or `timm` [40]. We divide each model into parts and report average score over certain part in order to enable comparison between the models as they do have different depths. Representations from the beginning until the global pooling are equally divided between *Early*, *Middle*, *Late* parts. *Pool* is the representation after the global pooling and *Final* is the output representation. We discriminate between the *Pool* part and the *Final* part to identify what role in achieving equivariance the global pooling and final classifier have. We measure translation equivariance over diagonal shifts of size 1 to 32 and rotation equivariance for $90°, 180°, 270°$ rotations.

Aside from the couple of state of the art CNNs we also measure the equivariance of the networks not relying directly on convolutional layers such as the ViT [13] and MLP-Mixer [38]. In case of CNNs the concept of a feature map is easily defined but in case of the others it may not be so clear. In ViT we first compute strided convolution with d output channels and stride 16. This results in d feature maps, where each "pixel" of size d is considered as a token. Next, such representation goes into 12 consecutive blocks consisting of a multi-head self-attention module, introduced by [39]. Self-attention computes the interaction between the tokens preserving the structure of the feature map therefore we can use our method at each layer of the network. The MLP-Mixer has very similar structure to the ViT, but instead of self-attention module, each block consists of the consecutive fully-connected linear layers. Each linear layer simply compute the mapping of each feature map separately and so we can identify feature maps in the same way as in ViT.

As for the equivariance property in these two architectures, the self-attention module in ViT is permutation equivariant, however, the division of the image into patches, the additional positional encoding that each patch gets and the linear layers can deteriorate the equivariance property. In MLP-Mixer, the self-attention is replaced by additional fully-connected linear layers, which by default do not have any equivariant properties.

4.3.1 Translation equivariance

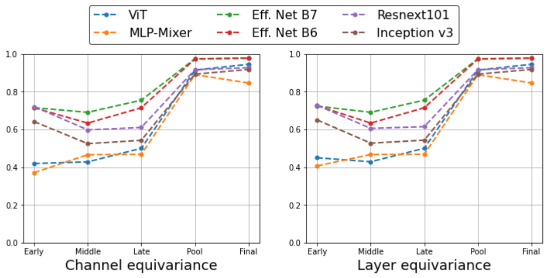


Figure 11: Channel (left) and layer (right) translation equivariance for different seminal models trained on ImageNet. CNNs exhibit more equivariance in the intermediate representation then models without strong equivariant prior like ViT or MLP-Mixer. Global pooling seems to play an important role in achieving invariance.

In Figure 11 we present the results for translation equivariance. We can see that ViT and MLP-Mixer have the least amount of translation equivariance although both of these models, in the Final part, achieve translation invariance comparable to CNNs. Lower translation equivariance can be the reason for the poor data efficiency of ViT and MLP-Mixer [13, 38] since we know that translation equivariance improves data-efficiency [13, 20]. We also see that EfficientNets have the highest amount of translation equivariance throughout all the layers, which may be the reason for their success among the convolutional architectures. Additionaly, we can observe that higher translation equivariance in throughout early, middle and late parts reults in higher invariance in the final layer. Finally, we see that layer and channel equivariance are the same meaning that these networks do not learn to represent different translations in different channels.

4.3.2 Rotation equivariance

In Figure 12 we report average equivariance for different parts of the seminal models. Again we can observe that the ViT and MLP-Mixer do not have a lot of channel or layer rotation equivariance although after the global pooling the ViT exhibits the most rotation invariance out of all the models. Secondly, we can see that the difference between channel and layer equivariance is the highest in the early parts of the network, particularly for Resnext101 and Inception v3. It also diminishes with the depth and in the late part becomes almost identical. In contrast to translation equivariance, we can see that models with low rotation equivariance throughout early, middle and late parts (ViT, Efficient-Net B6/B7) have the highest rotation invariance in the final part while model with highest equivariance in early, middle and late parts (Inception v3) has the least invariance in the final part. This can tell us that in case of learned rotation equivariance, high invariance in the prediction, doesn't have to result from high equivariance in the intermediate representations.

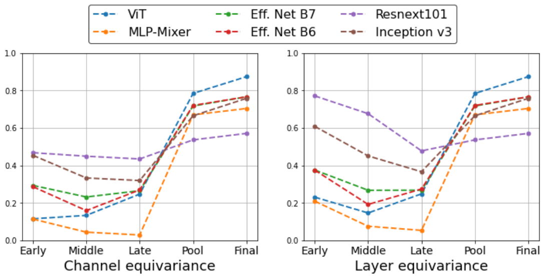


Figure 12: Channel (left) and layer (right) rotation equivariance for different seminal models trained on ImageNet. CNNs exhibit more equivariance in the intermediate representation then models without strong equivariant prior like ViT or MLP-Mixer. Early and Middle layer seem to have more layer than channel equivariance.

4.3.3 Does equivariance improve accuracy

In the final experiment we are interested what is the correlation between the validation accuracy and the amount of equivariance in the large scale models. For each part of the network we compute Spearman's rank correlation between the amount of equivariance and accuracy.

In Figure 13 we show the results for translation and rotation equivariance. Correlation between translation equivariance and accuracy grows steadily with the depth of the network, achieving almost perfect correlation in the final part. In case of rotation equivariance, there is no correlation between the equivariance in the representation before global pooling and the accuracy. Only after the global pooling the correlation becomes very high. This tells us that the invariance in the prediction is the most beneficial for higher accuracy. We also see that equivariance in the representation before global pooling is much more important for higher accuracy in case of embedded equivariance, ranging from 04 to 0.8, than learned equivariance, which oscilates around 0.1.

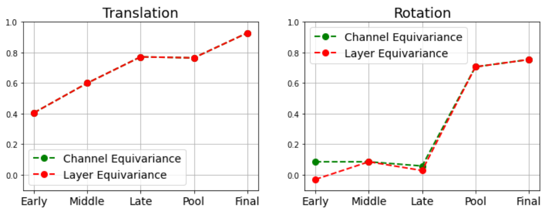


Figure 13: Correlation between the amount of translation (left) or rotation (right) equivariance and the accuracy for different parts of the network. Correlations for translation channel and layer equivariance are the same.

5. Conclusion

In this work, we conduct a study on learned equivariance in CNNs. First, we show why correlation may be a better option to compare feature maps than commonly used cosine similarity. Then, we explore how different factors of variation influence learning equivariant features. We show that imposing invariance or equivariance into the objective function does not induce learning more equivariant features and that equivariance or invariance in such cases is achieved in the final part of the network. We study how different data augmentation influence translation equivariance and find out that random crops and RandAugment increase translation equivariance of the network. We also show that smaller model capacity induces learning more translation equivariant features. Lastly, we quantify translation and rotation equivariance in different state of the art image classification models and showed that CNNs exhibit more translation and rotation equivariance than non-convolutional models like ViT or MLP-Mixer. We also find that only equivariance in the final part correlates highly with the classification accuracy.

Weaknesses and limitations One of the limitations of our method is that it allows to measure equivariance to affine transformations only. The reason for that is the transformation has to be defined in the $Z2$ space so that we can apply it on both the input image and the feature map. That disqualifies any transformation that for example changes the colors of the image.

Future work Applying equivariant priors usually adds additional cost in terms of memory or computation. A possible direction of future work could be to study whether one can apply equivariant priors only partially based on what the network learns. Another direction to explore is the relationship between learned equivariance and data efficiency.

References

[1] Anonymous. Do deep networks transfer invariances across classes? In *Submitted to The Tenth International Conference on Learning Representations*, 2022. under review. 2

[2] E. J. Bekkers. B-spline cnns on lie groups. *CoRR*, abs/1909.12057, 2019. 2

[3] E. J. Bekkers, M. W. Lafarge, M. Veta, K. A. J. Eppenhof, J. P. W. Pluim, and R. Duits. Roto-translation covariant convolutional networks for medical image analysis. *CoRR*, abs/1804.03393, 2018. 2

[4] Y. Bengio and Y. Lecun. Convolutional networks for images, speech, and time-series. 11 1997. 1

[5] D. Bouchacourt, M. Ibrahim, and A. S. Morcos. Grounding inductive biases in natural images:invariance stems from variations in data, 2021. 1, 2, 4

[6] J. Bruna and S. Mallat. Invariant scattering convolution networks, 2012. 2

[7] T. S. Cohen, M. Geiger, J. Köhler, and M. Welling. Spherical cnns. *CoRR*, abs/1801.10130, 2018. 2

[8] T. S. Cohen and M. Welling. Group equivariant convolutional networks. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, page 2990–2999. JMLR.org, 2016. 1, 2, 3, 6

[9] E. D. Cubuk, B. Zoph, J. Shlens, and Q. V. Le. Randaugment: Practical automated data augmentation with a reduced search space. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2020. 6

[10] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009. 2, 7

[11] L. Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012. 6

[12] S. Dieleman, J. De Fauw, and K. Kavukcuoglu. Exploiting cyclic symmetry in convolutional neural networks. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, page 1889–1898. JMLR.org, 2016. 1, 2

[13] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale, 2021. 1, 8

[14] C. Esteves, C. Allen-Blanchette, A. Makadia, and K. Daniilidis. 3d object classification and retrieval with spherical cnns. *CoRR*, abs/1711.06721, 2017. 2

[15] D. Freedman, R. Pisani, and R. Purves. Statistics (international student edition). *Pisani, R. Purves, 4th edn. WW Norton & Company, New York*, 2007. 4

[16] I. Goodfellow, Q. Le, A. Saxe, H. Lee, and A. Ng. Measuring invariances in deep networks. pages 646–654, 01 2009. 2

[17] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition, 2015. 4, 6

[18] M. A. Islam, M. Kowal, S. Jia, K. G. Derpanis, and N. D. B. Bruce. Global pooling, more than meets the eye: Position information is encoded channel-wise in cnns, 2021. 1, 2, 6

[19] E. Kauderer-Abrams. Quantifying translation-invariance in convolutional neural networks, 2017. 1, 2, 6, 7

[20] O. S. Kayhan and J. C. van Gemert. On translation invariance in cnns: Convolutional layers can exploit absolute spatial location, 2020. 1, 2, 3, 6, 8

[21] S. Kornblith, M. Norouzi, H. Lee, and G. Hinton. Similarity of neural network representations revisited, 2019. 4

[22] A. Krizhevsky. Learning multiple layers of features from tiny images. 2009. 4, 6

[23] M. Kuss. The geometry of kernel canonical correlation analysis. 06 2003. 4

[24] K. Lenc and A. Vedaldi. Understanding image representations by measuring their equivariance and equivalence. *CoRR*, abs/1411.5908, 2014. 1, 2

[25] J. C. Myburgh, C. Mouton, and M. H. Davel. Tracking translation invariance in cnns, 2021. 1, 2, 4

[26] C. Olah, N. Cammarata, L. Schubert, G. Goh, M. Petrov, and S. Carter. An overview of early vision in inceptionv1. *Distill*, 2020. https://distill.pub/2020/circuits/early-vision. 1, 2

[27] C. Olah, N. Cammarata, C. Voss, L. Schubert, and G. Goh. Naturally occurring equivariance in neural networks. *Distill*, 2020. https://distill.pub/2020/circuits/equivariance. 1, 2

[28] E. Oyallon and S. Mallat. Deep roto-translation scattering for object classification, 2015. 1, 2

[29] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. 7

[30] M. Raghu, J. Gilmer, J. Yosinski, and J. Sohl-Dickstein. Svcca: Singular vector canonical correlation analysis for deep learning dynamics and interpretability, 2017. 4

[31] D. W. Romero, E. J. Bekkers, J. M. Tomczak, and M. Hoogendoorn. Attentive group equivariant convolutional networks, 2020. 2

[32] D. W. Romero and J.-B. Cordonnier. Group equivariant stand-alone self-attention for vision, 2021. 2

[33] D. W. Romero and M. Hoogendoorn. Co-attentive equivariant neural networks: Focusing equivariance on transformations co-occurring in data, 2020. 2

[34] D. W. Romero and S. Lohit. Learning equivariances and partial equivariances from data, 2021. 2

[35] A. Ruderman, N. C. Rabinowitz, A. S. Morcos, and D. Zoran. Pooling is neither necessary nor sufficient for appropriate deformation stability in cnns, 2018. 2

[36] I. Sosnovik, M. Szmaja, and A. Smeulders. Scale-equivariant steerable networks, 2020. 1, 2

[37] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions, 2014. 2

[38] I. Tolstikhin, N. Houlsby, A. Kolesnikov, L. Beyer, X. Zhai, T. Unterthiner, J. Yung, A. Steiner, D. Keysers, J. Uszkoreit, M. Lucic, and A. Dosovitskiy. Mlp-mixer: An all-mlp architecture for vision, 2021. 8

[39] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017. 8

[40] R. Wightman. Pytorch image models. `https://github.com/rwightman/pytorch-image-models`, 2019. 7

[41] D. E. Worrall, S. J. Garbin, D. Turmukhambetov, and G. J. Brostow. Harmonic networks: Deep translation and rotation equivariance, 2017. 2

[42] S. Yun, D. Han, S. J. Oh, S. Chun, J. Choe, and Y. Yoo. Cutmix: Regularization strategy to train strong classifiers with localizable features, 2019. 6

[43] S. Zagoruyko and N. Komodakis. Wide residual networks, 2017. 7

[44] R. Zhang. Making convolutional networks shift-invariant again. *CoRR*, abs/1904.11486, 2019. 1, 2, 3, 4

[45] A. Zhou, T. Knowles, and C. Finn. Meta-learning symmetries by reparameterization, 2021. 2

6. Appendix

6.1. Training settings

Parameter	Value
Epochs	100
Learning rate	0.01
Optimizer	Adam
L2 reg.	0.0005
Weight Decay epochs	25, 50
Weight Decay factor	10.0
Batch size	128

Table 1: CNN training parameters.

Parameter	Value
Epochs	200
Learning rate	0.1
Optimizer	SGD
Momentum	0.9
L2 reg.	0.0001
Weight Decay epochs	100, 150
Weight Decay factor	10.0
Batch size	128

Table 2: Resnet44 training parameters.

2

Background on Deep Learning

2.1. What is Deep Learning?

The success of Machine Learning techniques was due to their ability to acquire knowledge by extracting patterns from raw data. However, the performance of these algorithms depends heavily on the representation of the data they are presented with. Such an algorithm can, for example, discover the correlation of certain data features with different outcomes, however, it cannot create features themselves. This burden lies on the person who uses the algorithm.

The solution to this problem is to use an algorithm that, by itself, can discover the features from the data. Such an approach is called representational learning. Deep Learning is a representation learning technique that learns to represent complex features by combining different, simpler, learned features.

2.2. Deep Neural Networks

2.2.1. Perceptron.

The basic building block of Neural Network is the algorithm called Perceptron [1]. It's a very simple algorithm that works as follows. Given a sample vector $x \in R^d$, we define learnable parameters $W \in R^d$, $b \in R$. In a forward pass we compute $y = \sigma(W * x + b) \in R$, where σ is a certain non-linearity function. Such process is presented in Figure 2.1. It is usually referred to as a single neuron.

2.2.2. Multi-Layer Perceptron

The main idea of neural networks is to stack single perceptrons together. Firstly, we can stack the neurons horizontally to obtain a Single Layer Neural Network. The formulation barely changes, but now given a sample vector $x \in R^d$, the learnable parameters are $W \in R^{d \times n}$, $b \in R^n$ and the output $y \in R^n$, where n is the number of horizontally stacked neurons also called the width of a layer.

At this point, there is nothing deep about Single Layer Network. To obtain a Deep Neural Network we have to stack Single layer Neural Networks on top of each other.

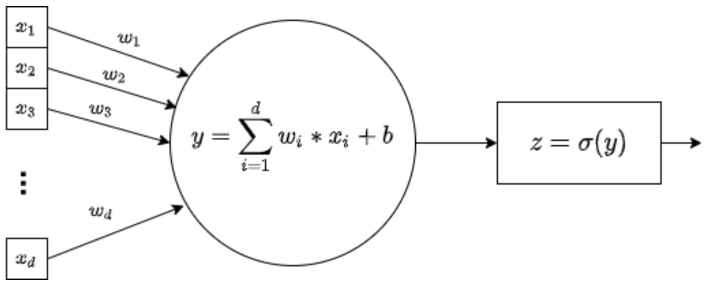


Figure 2.1: Single Perceptron.

For example, we might have three functions (layers) $f(1), f(2)$, and $f(3)$ connected in a chain, to form:

$$f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x))) \tag{2.1}$$

In this case, $f^{(1)}$ is called the first layer of the network, $f^{(2)}$ is called the second layer, and $f^{(3)}$ is the output layer of the network. Such a network is called Multi-Layer Perceptron (MLP).

2.2.3. ACTIVATION FUNCTION

The crucial part of the construction of the single neuron are non-linearities, also called activation functions. If we were to throw away the non-linearity from each neuron, the whole network becomes a linear model and, regardless of the number of layers, could be represented as $y = \boldsymbol{W} * \boldsymbol{x}$. The power of such a linear model is significantly limited as it requires the target classes to be linearly separable to be effective. There are many possible choices of activation functions. Among the most popular are:

- **Sigmoid** - One of the earliest activation functions was the Sigmoid function. However, it suffers from the dying gradient problem. Input values above 5 and below -5 results in the same value 2.2, which results in very small gradients. That's why the Sigmoid is very sensitive to network initialization.

$$f(x) = \frac{1}{1+e^{-x}} \tag{2.2}$$

- **tanh** - Another function used in the 'early days' is hyperbolic tangent. It has a similar characteristic to Sigmoid but it has the advantage that it allows also for negative values. Similarly to Sigmoid, it suffers from dying gradient problem on both ends.

$$f(x) = \frac{sinh}{cosh} \tag{2.3}$$

- **ReLU** - Rectified Linear Unit (ReLU) is currently the most widely used activation function in Deep Learning for Computer Vision. However, it still suffers from the dying gradient problem for input values below 0.

$$f(x) = max(0, x) \tag{2.4}$$

- **LeakyRelu** - To alleviate ReLU's dying gradient issue, Leaky ReLU allows for small negative values.

$$f(x) = \begin{cases} 0.01x, & \text{if } x < 0. \\ x, & \text{otherwise.} \end{cases} \tag{2.5}$$

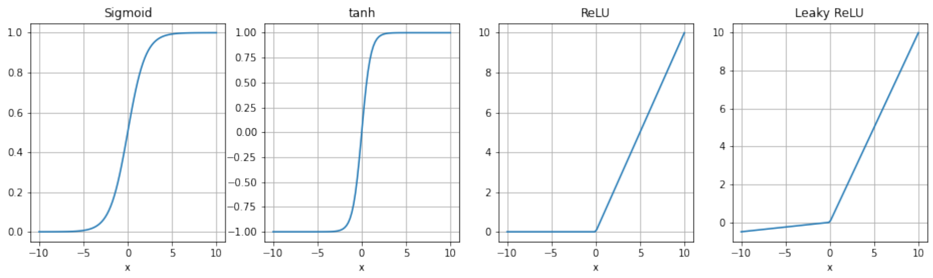


Figure 2.2: Different activation functions.

2.3. Convolutional Neural Networks

MLPs, although very powerful, have some limitations in their basic form. In the presence of sensory data such as images, video, or audio the spatio-temporal context is introduced in the data. What that means is that in our data we can have the same reoccurring patterns but in a different time or spatial locations. The MLP is by design not aware of the existence of a spatio-temporal context and thus has to learn different parameters to learn the same patterns in different locations. To alleviate this issue [2] introduced a translation equivariance property into the MLP and proposed the Convolutional Neural Networks.

In a single layer of the CNN, each neuron, instead of simply computing $y = W * x + b$, computes a discrete convolution

$$y(i,j) = (x \star W)(i,j) + b = \left(\sum_m \sum_n x[i+m, j+n] * W[m,n] \right) + b \tag{2.6}$$

, where the learnable parameters W are so-called filters or kernels. In case of images, discrete convolution is equivalent to sliding the filter over every location in the image and at each location computing the dot-product with the local neighbourhood. This process is depicted in Figure 2.3. Learnable weights can be seen as feature detectors, where by sliding the filter over the image we are essentially detecting the presence of a certain feature (Figure 2.4).

In a multilayer CNN, a typical layer of a convolutional network consists of three stages. In the first stage, the input of the layer is convolved with multiple filters, which results in a stack of feature maps, where a single feature map represents a presence of a certain feature. In the second stage, each linear activation is run through a nonlinear activation function, such as the ReLU function. In the third stage, a pooling function is used to reduce the spatial extent of the feature maps (Figure 2.5).

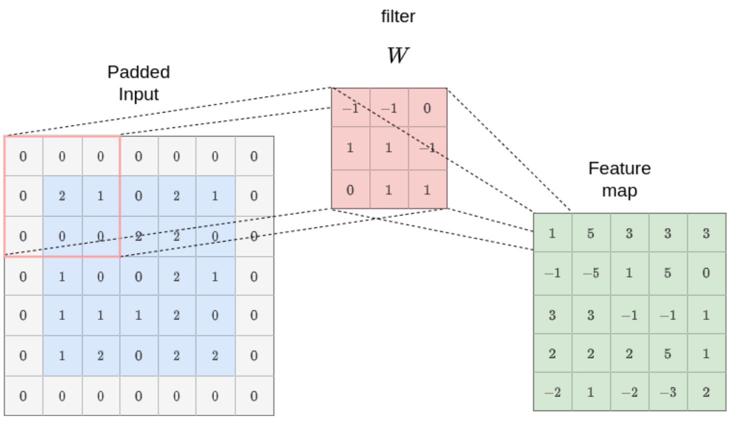


Figure 2.3: 2D discrete convolution. For every position (x,y) in the image, a dot-product is computed between the filter W and the local neighborhood of (x,y).

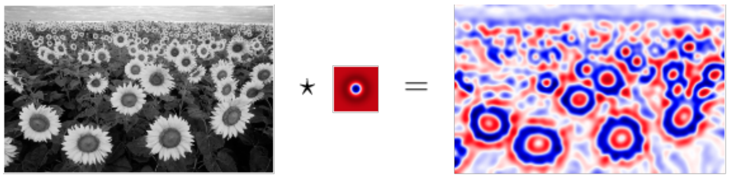


Figure 2.4: Convolution serves as a feature detector. Learning weights can be interpreted as learning features.

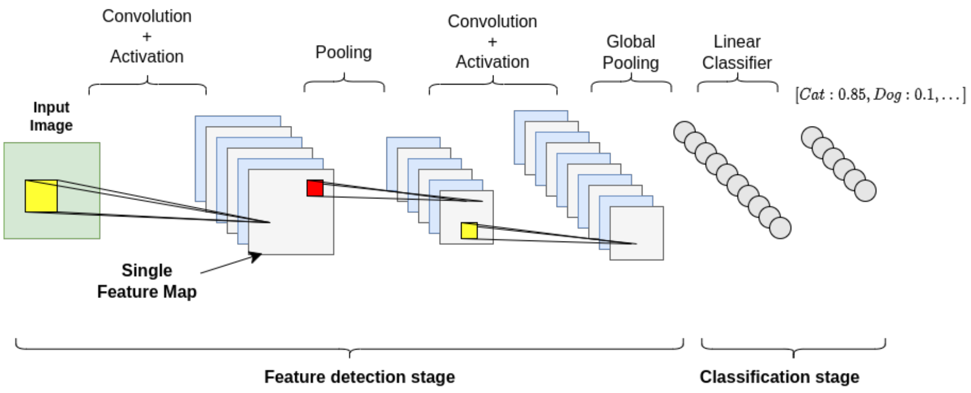


Figure 2.5: A simple visualization of a CNN. Convolutional layer results in the stack of feature maps and pooling layer reduces the spatial size of these feature maps. Usually at the end a global pooling is computed, which reduces each feature map to a single value.

2.3.1. POOLING

Pooling function replaces the output of the network at a certain spatial location with a summary statistic of the nearby outputs (Figure 2.6). In general, such an operation makes the representation invariant to local transformations. There are many examples of different pooling functions:

- **Average-pooling** - Computes the average of the local neighborhood.
- **Max-pooling** - Computes the maximum of the local neighborhood.
- **Strided Convolution** - Instead of directly encoding what operation over the local neighbourhood should be computed into architecture, we can leave it for the network to learn by itself using strided convolution. In strided convolution, instead of shifting the filter by 1 pixel, we simply shift it by multiple pixels, which results in reduced size of the output. The filters, which determine the aggregating function, are learned by the network [3].
- **Blur-pooling** - The sub-sampling is preceded by the gaussian blurring to improve the translation equivariance property, which deteriorates due to sub-sampling [4].

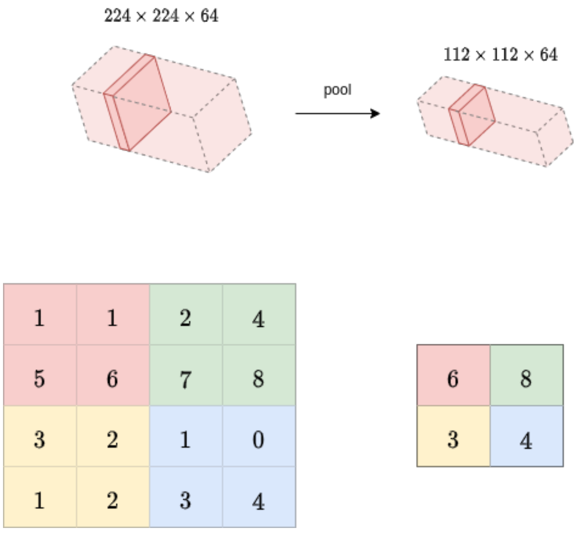


Figure 2.6: Pooling operation. A certain aggregating function is computed over a local neighborhood, reducing the spatial size of a feature map.

2.4. How to train neural networks

The difference between the linear models and neural networks is that the non-linearity of a neural network causes the loss functions to become non-convex. That means the neural networks are usually trained by using iterative, gradient-based optimizers that, instead of using convex optimization solvers as is the case with e.g. SVMs, just drive the cost function to a very low value. That means that we iteratively update the model's parameters in order to minimize the cost function. The process of updating the parameters can be divided into three parts: forward pass, backpropagation, updating the parameters.

2.4.1. FORWARD PASS

In the forward pass, the input is passed through the network, and the cost function is computed. In the case of the classification problem output of the network is a vector $y \in R^c$, where c is the number of classes. Each element of that vector denotes the probability of a particular class. The cost function in classification tasks is most commonly the Cross-Entropy Loss:

$$J(\Theta) = -(t * log(y)) + (1 - t) * log(1 - y), \tag{2.7}$$

where t is the ground truth and y is the output of the network. Having the value of the cost function we can now proceed to compute the gradients.

2.4.2. BACK PROPAGATION

Once we have the value of the cost function, computing the gradient of the parameters $\frac{\partial J(\Theta)}{\partial \Theta}$ may seem straightforward. However, considering the depth and the number of parameters in the neural network, it would be computationally expensive to compute the gradient of every single parameter separately. That's where the backpropagation algorithm [5], often simply called backprop, comes to use. Backprop uses the chain rule of calculating the derivatives of nested functions. In a simple univariate case the chain rule looks as follows:

$$\frac{d}{dx} f(g(x)) = \frac{\partial f}{\partial g} \frac{dg}{dx} \tag{2.8}$$

Such a rule allows us to reuse gradient computed at layer $l + 1$ to compute gradients at layer l.

2.4.3. UPDATING THE PARAMETERS

By updating the parameters using the gradient, we gradually move the cost function towards local or global minimum. A very important part of this process is the learning rate ϵ, which determines how big are the updates of the learning parameters. There are different algorithms for updating the parameters, which are commonly called optimizers:

- **Gradient Descent** - Gradient descent algorithm is the basis of most of the optimizers used in Deep Learning. First we compute the gradient for all the samples in the training dataset:

  $$\hat{g} \leftarrow \frac{1}{N} \nabla_\theta \sum_i J(f(x^{(i)};\theta)) y^i) \tag{2.9}$$

  and then update the parameters using the fixed learning rate ϵ:

  $$\theta \leftarrow \theta - \epsilon \hat{g} \tag{2.10}$$

- **Stochastic Gradient Descent (SGD)** [6] - It is often the case that we cannot process our whole dataset in one go and have to do this in a batched manner. In such a case, we are computing the gradient and updating the weights based on randomly sampled m training examples. This variation is called Stochastic Gradient Descent (SGD). Sampling m training examples introduce noise into our optimization process that does not disappear even when we arrive at a minimum. To mitigate that, it is necessary to gradually decrease the learning rate over time. Now at iteration k, the algorithm is as follows:

$$\begin{aligned} &\hat{g} \leftarrow \frac{1}{m}\nabla_\theta \sum_{i=1}^{m} J(f(x^{(i)};\theta))y^{i}) \\ &\theta \leftarrow \theta - \epsilon_k \hat{g} \end{aligned} \tag{2.11}$$

- **SGD with Momentum** - One way to deal with the noisy gradients is to smooth out the average gradient. In this case, we will not use the direct gradient to update the parameters but instead, we will use the exponentially weighted moving average of the gradients.

$$\begin{aligned} &\hat{v}_k \leftarrow \hat{v}_{k-1}\rho + (1-\rho)\frac{1}{m}\nabla_\theta \sum_{i=1}^{m} J(f(x^{(i)};\theta))y^{i}) \\ &\theta \leftarrow \theta - \epsilon_k \hat{v}_k \end{aligned} \tag{2.12}$$

- **SGD with RMSProp** - The momentum only makes the mean gradient change smoothly, however, the mean does not tell the full story. It can happen that the variance of the gradients is very high in the directions orthogonal to the direction of the convergence of the mean gradient. So, even though the mean is converging in a certain direction it can still do so in a noisy way.

$$\begin{aligned} &\hat{r}_k \leftarrow \hat{r}_{k-1}\rho + (1-\rho)\frac{1}{m}\nabla_\theta^2 \sum_{i=1}^{m} J(f(x^{(i)};\theta))y^{i}) \\ &\theta \leftarrow \theta - \epsilon_k \frac{\nabla_\theta}{\sqrt{\hat{r}_k}} \end{aligned} \tag{2.13}$$

- **SGD with Adam** [7] - Adaptive Moment Estimation (Adam) combines Momentum with RMSProp.

$$\begin{aligned} &v_k \leftarrow \hat{v}_{k-1}\rho_1 + (1-\rho_1)\frac{1}{m}\nabla_\theta && r_k \leftarrow \hat{r}_{k-1}\rho_2 + (1-\rho_2)\frac{1}{m}\nabla_\theta^2 \\ &\hat{v}_k = \frac{v_k}{1-\rho_1^k} && \hat{r}_k = \frac{r_k}{1-\rho_2^k} \\ &\theta \leftarrow \theta - \epsilon_k \frac{\hat{v}_k}{\sqrt{\hat{r}_k}} \end{aligned}$$

2.5. TUNING NEURAL NETWORKS

Having the model and the data is not enough to obtain a properly working algorithm. Similarly to other machine learning models, neural networks need to be trained properly. The main challenge in training machine learning models is that our algorithm has to perform well not only on the data samples it was trained on but also on a new, previously unseen data. The ability to perform well on previously unobserved samples is called generalization. Training a machine learning model is usually done on a training set for which we can compute training error and throughout the training we aim to reduce it. However, the performance of a model is typically evaluated not by it's training error but rather a test error, which we can also be called a generalization error. This test error is the performance of the model on a test set consisting of examples that were collected separately from the training set. The factors determining how well a machine learning algorithm will perform are its ability to: a) Make the training error small; b) Make the gap between training and test error small. These requirements correspond to 2 main challenges when training machine learning models:

- **Underfitting** - Occurs when both the training error and the test error are too high. It usually means that our model is too simple and therefore cannot create complex enough representation to discriminate between classes.
- **Overfitting** - Occurs when the training error is small but the test error is high. Most often the reason for that is our model being too complex, which facilitates simply remembering the training dataset instead of learning more general representation.

2.5.1. Regularization

Regularization, generally speaking, can be seen as any modification in our model or the training process, that leads to smaller generalization error but not training error. These are the common regularization techniques used in training CNNs.

Penalty in the objective

Commonly in machine learning models, one can add an additional term to the cost function that penalizes certain behaviours of the model. One of the most common penalizing terms are:

- **L2-norm of the weights** - This term makes the model prefer weights with the smaller L2-norm, where parameter λ controls how punishing we want the additional term to be.

$$\Omega(\theta) = \lambda\theta^T\theta \tag{2.14}$$

- **L1-norm of the weights** - The idea here is similar as in case of the L2-regularization, however, in this case we get the additional property of feature selection.

$$\Omega(\theta) = \lambda|\theta| \tag{2.15}$$

Early stopping

Early stopping is quite simple idea to prevent overfitting. When training the model, aside from giving the model access to the training set we also give it access to so called validation set. During training, we additionally evaluate the model's performance on the validation set. After observing that the validation error start to increase while the training error is decreasing we simply stop the training. This way we hope that the model will generalize on the unseen test set.

DROPOUT

Dropout [8] is a regularization technique specific to neural networks. During each training step, dropout randomly select fraction p of all the neurons and mutes their output. This process can be repeated every batch or every epoch and leads to significantly lower generalization error rates, as the presence of neurons is made unreliable. It can also be seen as training an ensemble of models.

During testings we cannot just randomly turn off neurons and so what is being done is that we retain all the activations but the output is scaled by $1-p$.

BATCH NORMALIZATION

When dealing with machine learning problems, we assume that our data, both training and testing, follow a distribution that is constant for a particular task. That is quite an important requirement since a discrepancy between training and testing distribution can lead to poor generalization. A similar argument can be made with regards not only to the input data but also to the intermediate representations. However, because throughout the training the weights are constantly updated, the input distribution of the intermediate layers is constantly changing. This process is known as an internal covariate shift. It makes the training slow, requires a small learning rate and a good parameter initialization, which makes the network difficult to train. This problem is alleviated by normalizing the layer's inputs over a mini-batch. This process is therefore called Batch Normalization [9]. It work as follows:

1. Calculate the mean of the mini-batch for output feature k of every neuron

$$\mu_b^{(k)} = \frac{1}{m}\sum_{i=0}^{m} x_i^{(k)} \tag{2.16}$$

2. Calculate the variance of the mini batch

$$\sigma_b^{2(k)} = \frac{1}{m}\sum_{i=0}^{m}(x_i^{(k)} - \mu_b^{(k)})^2 \tag{2.17}$$

3. Normalize the representation to 0 mean and unit variance

$$\hat{x}_i{}^{(k)} = \frac{x_i^{(k)} - \mu_B^{(k)}}{\sqrt{\sigma_B^{2(k)}\epsilon}} \tag{2.18}$$

4. Scale and shift the normalized mini-batch to set the new mean and variance.

$$y_i = \gamma x_i + \beta \tag{2.19}$$

DATA AUGMENTATION

Another common way of improving the generalization is data augmentation. Data augmentation is a technique of applying transformations, usually randomly parametrized, to the input of the neural network. Such a process incentives a network to learn invariance to different input transformations. In Figure 2.7 we show examples of very common data augmentations.

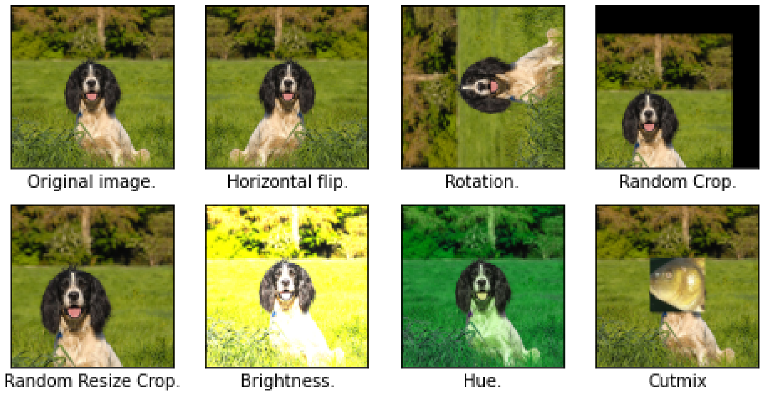


Figure 2.7: Examples of common data augmentation techniques.

REFERENCES

[1] S. Shalev-Shwartz, *Perceptron algorithm,* in *Encyclopedia of Algorithms*, edited by M.-Y. Kao (Springer US, Boston, MA, 2008) pp. 642–644.

[2] Y. Bengio and Y. Lecun, *Convolutional networks for images, speech, and time-series,* (1997).

[3] K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition,* (2015), arXiv:1512.03385 [cs.CV] .

[4] R. Zhang, *Making convolutional networks shift-invariant again,* CoRR **abs/1904.11486** (2019), arXiv:1904.11486 .

[5] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Learning representations by back-propagating errors,* Nature **323**, 533 (1986).

[6] S. Ruder, *An overview of gradient descent optimization algorithms,* arXiv preprint arXiv:1609.04747 (2016).

[7] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization,* CoRR **abs/1412.6980** (2015).

[8] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, *Dropout: A simple way to prevent neural networks from overfitting,* J. Mach. Learn. Res. **15**, 1929–1958 (2014).

[9] S. Ioffe and C. Szegedy, *Batch normalization: Accelerating deep network training by reducing internal covariate shift,* CoRR **abs/1502.03167** (2015), 1502.03167 .

3 Equivariance In Convolutional Neural Networks

3.1. Translation equivariance in Convolutional Neural Networks

Function $f : X \Rightarrow Y$ is commonly considered to be equivariant with respect to transformations T if it commutes with this transformation:

$$f(T(x)) = T(f(x)) \quad \forall x \in X. \tag{3.1}$$

In other words, it means that we can apply a transformation to the input or to the output and obtain the same result. Convolution is translation equivariant, which is depicted in Figure 3.1.

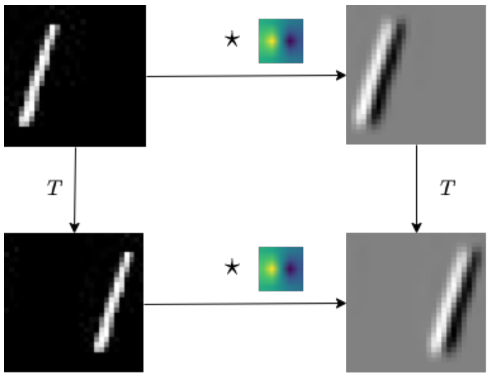


Figure 3.1: Translation equivariance property of the convolution. One can either translate the image and then convolve it with the filter or first convolve the image and then translate the output.

3.1.1. Violating translation equivariance

There are several factors that deprecate the translation equivariance property of the convolutional neural networks.

Sampling

In [1] the author shows how different forms of subsampling currently used in CNNs like max-pooling and strided convolution can deteriorate translation equivariance property. In principle, he shows that these methods introduce heavy aliasing, which can result in large output changes for the small input changes. To alleviate this issue, he precedes the subsampling operation with a blurring operation reducing the aliasing effect and improving the translation equivariance property (Figure 3.1).

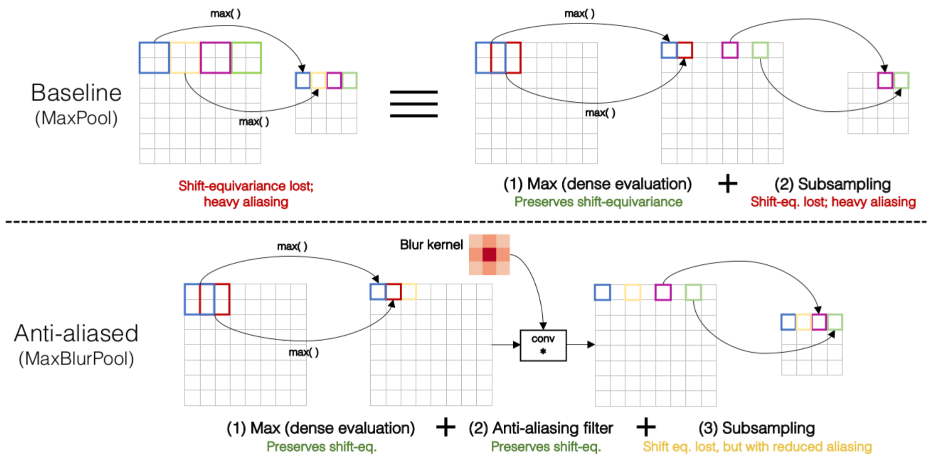


Figure 3.2: Anti-aliassing BlurPool. Pooling does not preserve translation equivariance. It is functionally equivalent to densely-evaluated pooling, followed by subsampling, which loses translation equivariance. (Bottom) Bluring filter is applied before subsampling. This keeps the first operation unchanged, while anti-aliasing the appropriate signal. Figure courtesy of [1].

Border effects

In [2] the authors show how CNNs exploit the border effects to classify objects that are close to the border, therefore, breaking the translation equivariance. The *Same Convolution*, which is the mode of the convolution that adds padding to retain the output image size, has been the most common way to apply convolution in CNNs. However, this mode allows for the scenarios where we have a filter detecting a certain feature and depending on at which border the feature is located in the image, it can be detected or not. To alleviate this issue, the authors propose *Full Convolution*, which adds more padding to enable the filter detect features at every position in the image. (Fig. 3.3)

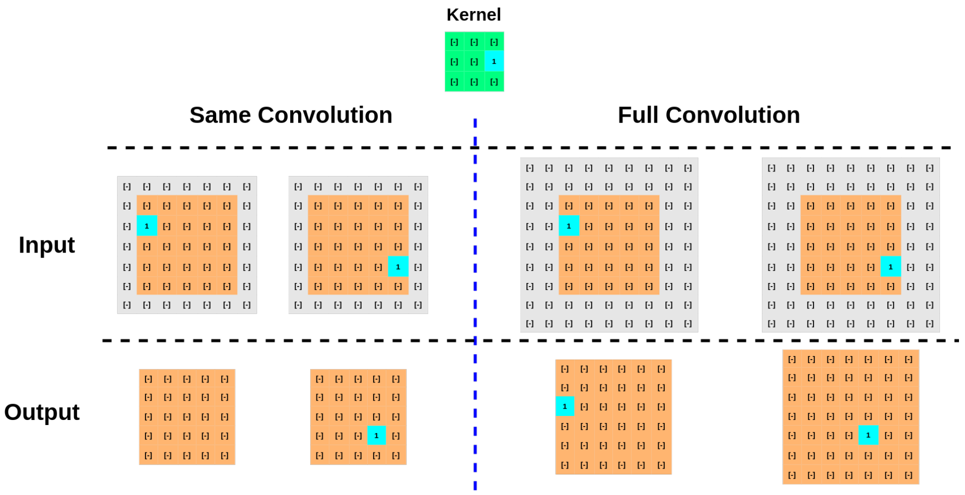


Figure 3.3: Lack of translation equivariance due to the border effect. Blue square in the input represents presence of a certain feature and blue square in the output represents activation to that features. Same Convolution adds not enough padding to capture features located at the border. Full Convolution solves that by adding extra padding.

3.2. EQUIVARIANCE TO OTHER TRANSFORMATIONS

Convolution operation is not equivariant to other affine transformations e.g. rotations (Figure 3.4). There are number of works that introduce the equivariance property over different transformations into the CNNs. Most notable is probably the work of [3], where the authors show how to make convolution operation equivariant to any discrete transformation symmetry group. Let's analyze the mechanism behind such method as an example of how one can achieve equivariance.

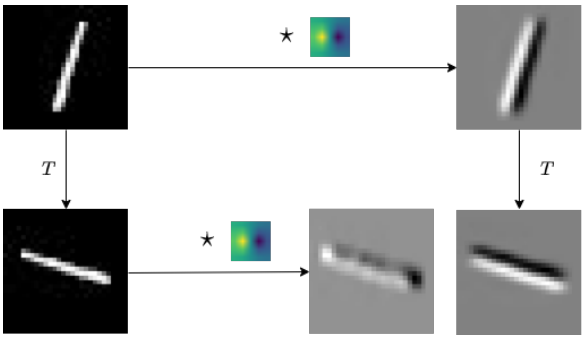


Figure 3.4: Lack of rotation equivariance in the convolution operation.

3.2.1. Group Equivariant Convolutional Networks

So far we've defined equivariance as a property of a function, where domain and codomain commute with a particular transformation T. Here we need to introduce a subtle change to our definition. More formally equivairance is the property of the function under a transformation group G if it's domain and codomain commute with the action of that group:

$$f(T_g(x)) = T_g(f(x)) \quad \forall (x, g) \in (X, G). \tag{3.2}$$

Group Equivariant Convolution (GConv), proposed in [3], allows making the convolution operation equivariant to any transformation symmetry group. To explain how the GConv works, we introduce some of the basic concepts of group theory.

Symmetry transformation & Symmetry group

First, let's define what a symmetry transformation is. Symmetry transformation is a transformation that leaves the object invariant, meaning that the parts of the objects might be permuted, but the object as a whole remains unchanged. An example of such transformation can be a 90° rotation or a horizontal flip.

A symmetry group G is a set of symmetry transformations where for any transformations g, h and $i \in G$ the following conditions hold:

1. Composing g and h gives another symmetry transformation gh, which also belongs to the group G $\forall g, h \in G$.
2. The inverse transformation g^{-1} of any symmetry transformation is also symmetry transformation and composing it with g give the identity transformation $\forall g \in G$.
3. Elements of the group are associative meaning $i(gh) = (ig)h \quad \forall i, g, h \in G$.
4. There exists an element $e \in G$ for which $ge = eg = g \quad \forall g \in G$.

An example of such group is a set of 2D integer translations Z^2, under which the regular convolution is already equivariant. One can verify that for any tuples of integers (n, m), where n, m is the size of translation in x and y directions, the conditions stated above hold. However, we want to get more than that. Here we show how we can make the convolution equivariant under the $p4$ group, which consists of all compositions of translations and rotations by 90 degrees. The rotations in $p4$ group is visualized in Figure 3.5.

Group Equivariance

Having a basic notion of symmetry group and an example in the form of $p4$ group, let's analyze what properties the convolutions needs to have in order to be equivariant with respect to the transformation symmetry group. We are concerned with rotations of the 2D images, so we introduce the following notation for a rotation r acting on a 2D grid.

$$[T_r I](x) = I(r^{-1}x) \tag{3.3}$$

This says that to get the value of the r rotated image $T_r I$ at the point x, we need to do a lookup in the original image I at the point $r^{-1}x$, which is the unique point that gets mapped to x by g.

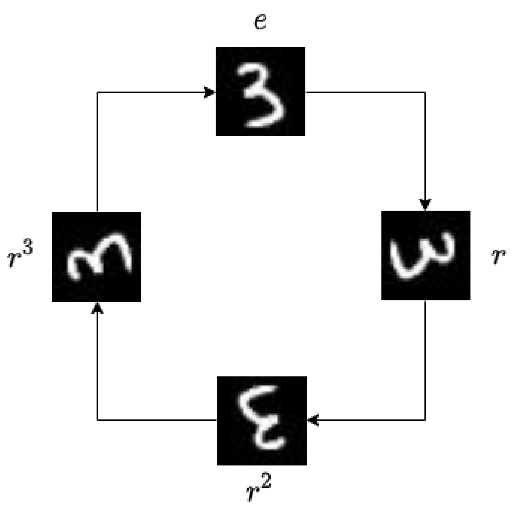


Figure 3.5: Visualization of the $p4$ group.

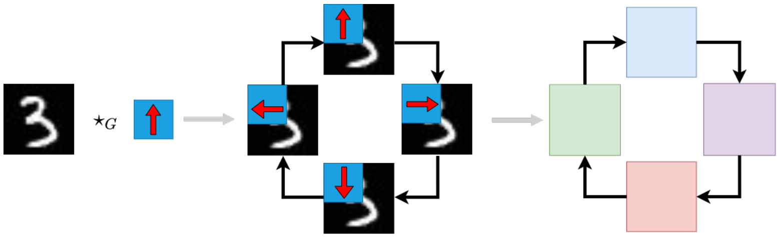
Here, we show the derivation of the necessary conditions for the convolution to be equivariant under a group action, originally done by [3].

$$\begin{aligned}
[[T_r I] \star \psi](x) &= \sum_{y \in Z^2} \sum_k T_r f_k(y) \psi(y - x) \\
&= \sum_{y \in Z^2} \sum_k f_k(r^{-1} y) \psi(y - x) \\
&= \sum_{y \in Z^2} \sum_k f_k(y) \psi(r y - x) \\
&= \sum_{y \in Z^2} \sum_k f_k(y) \psi(r(y - r^{-1} x)) \\
&= \sum_{y \in Z^2} \sum_k f_k(y) T_{r^{-1}} \psi(y - r^{-1} x) \\
&= f \star [T_{r^{-1}} \psi](r^{-1} x) \\
&= T_r [f \star [T_{r^{-1}} \psi]](x)
\end{aligned} \tag{3.4}$$

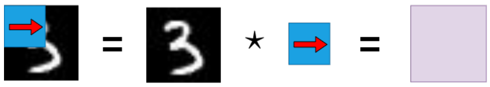
What the Equation 3.4 says is that the convolution of a rotated image $T_r I$ with a filter ψ is the same as rotating by r the image I convolved with the inverse-rotated filter $T_{r^{-1}} \psi$. It means that if the single filter is rotation invariant $\psi = T_{r^{-1}} \psi$, the convolution is equivariant or if CNN learns rotated copies of the same filter, the stack of feature maps is equivariant, although individual feature maps are not.

GROUP EQUIVARIANT CONVOLUTION

For the convolution to be equivatiant to the symmetry group we have to compute convolution for each element of the group. That means, instead of computing single convolution using filter ψ, we compute four convolutions with rotated versions of the filter ψ (Figure 3.6). This is so-called $P4Z2$ convolution. An important observation here is that, although the input and the filter are functions on the plane Z^2, the output of such convolution is a function on $p4$ group. The consequence of that is we cannot apply the same $P4Z2$ convolution in the subsequent convolutional layer since the input to that layer is a function on $p4$ group.



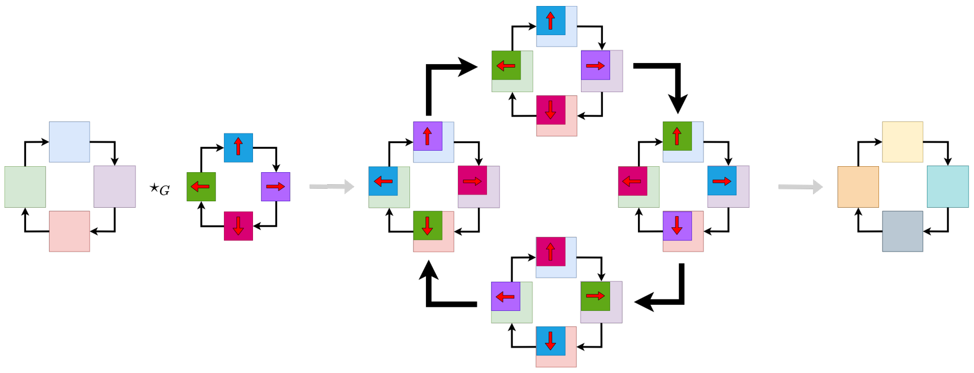
(a) P4Z2 convolution. In a single group convolution operation, four convolution are computed with filter transformed by each element of the group.



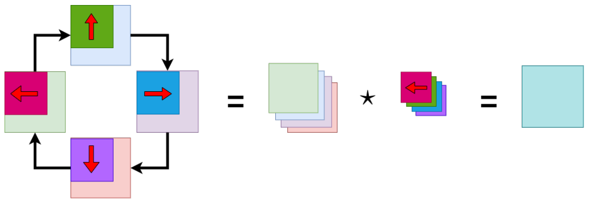
(b) For single group element, a regular convolution is computed.

Figure 3.6: P4Z2 convolution.

Since the input to the subsequent layer is the function on $p4$, the filter has to be function on $p4$ as well. This adds one caveat to transformation of the filter. In case of the $P4Z2$ convolution we were simply rotating the filter. Now that our input is in $p4$ space and hence the filter is, the transformation looks a bit different. Apart from rotating the filter, elements of the filters are also permuted (Figure 3.7). This way, we can extend the equivariance property to the subsequent layers and build fully equivariant networks.



(a) P4P4 convolution. Transformation of the filter defined in $p4$ space is now not only the rotation but also a permutation of individual channels.



(b) For single group element, a regular convolution is computed. The input and the filter have 4 channels now.

Figure 3.7: P4P4 convolution.

REFERENCES

[1] R. Zhang, *Making convolutional networks shift-invariant again,* CoRR **abs/1904.11486** (2019), arXiv:1904.11486 .

[2] O. S. Kayhan and J. C. van Gemert, *On translation invariance in cnns: Convolutional layers can exploit absolute spatial location,* (2020), arXiv:2003.07064 [cs.CV] .

[3] T. S. Cohen and M. Welling, *Group equivariant convolutional networks,* in *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16 (JMLR.org, 2016) p. 2990–2999.