Invariance and Equivariance Induced Weight Sharing Schemes

Anonymous Author(s)
Affiliation
Address
email

Abstract

The present-day method of deep neural network design is governed largely by empiricism and intuition. Domain knowledge of the training data is inserted into model through clever techniques such as data augmentation and learned embeddings, but for the most part is left unexploited. Furthermore, today's networks can have millions of parameters with a large amount of redundancy, risking overfitting and increasing training time. This paper proposes a systematic procedure for integrating domain knowledge of the data into the design of the neural network itself. This is accomplished by first identifying transformations a representation should be invariant or equivariant under. Then, these transformations are used to constrain the representation's weights in such a way that the invariance or equivariance is guaranteed to be preserved. It is shown that the weight constraints implied by both recurrent neural networks and convolutional neural networks can be interpreted as special cases of this procedure. A few extensions of this procedure are covered, as well as a proof of concept exercise using the AlexNet architecture.

1 Introduction

The success in recent years of deep neural networks (DNN) has been far reaching and pervasive. Their architecture allows for complex representations to be built from largely unstructured data such as images [5], natural language [11], and video [4] to name a few. These expressive representations are possible in part because of the large number of parameters that make up the models. Some of the most successful models have a nearly impractical number of parameters given today's computing power. NASNet, the current ImageNet accuracy leader has over 10 million free parameters [12]. VGGNet has over 100 million parameters and takes 2 to 3 weeks to train on several GPUs [9]. We do expect computing power to make this scale of parameters more accessible and easier to work with, but this will also most likely lead to even larger networks. Furthermore, the risk of overfitting tends to increase with scale. Various methods such as dropout [10] and batch normalization [3] have been successful in alleviating some of the ill effects of large scale networks, but in general these techniques are best used in conjunction with attempts to limit the number of parameters.

A generalized method to create collections of weights that are constrained to match some property of the input and output representations of a neural network layer is proposed. The properties discussed in this paper are limited to invariance and equivariance but there is no reason the same idea could not be applied to other types of relationships between representations.

1.1 Network Layer

A typical layer in a neural network $f : X \to Y$ with $X \subset \mathbb{R}^{m_1 \times \cdots \times m_M}$ and $Y \subset \mathbb{R}^{n_1 \times \cdots \times n_N}$ can be decomposed as follows:

Submitted to 32nd Conference on Neural Information Processing Systems (NIPS 2018). Do not distribute.

$$\begin{aligned} f(x) &= \sigma \circ A_b \circ L_w(x) \quad x \in X \\ A_b(y) &= y + b \quad y, b \in Y \end{aligned} \tag{1}$$

$\sigma : Y \to Y$ is to be interpreted as an activation function, such as relu, tanh, etc., and $A_b : Y \to Y$ is a translation. x, y, and b are all tensors and can be written in an indexed notation: $x_{i_1 \ldots i_M}$, $y_{j_1 \ldots j_N}, b_{j_1 \ldots j_N}$. $L_w : X \to Y$ is a linear transformation defined by the tensor $w \in W \subset \mathbb{R}^{n_1 \times \cdots \times n_N \times m_1 \times \cdots \times m_M}$, which can also be written in an indexed notation:

$$L_w(x)_{j_1 \ldots j_N} = \sum_{i_1, \ldots, i_M} w_{j_1 \ldots j_N i_1 \ldots i_M} x_{i_1 \ldots i_M} \tag{2}$$

A simple example of this would be the first layer of a network, which takes images of size 1600×1000 as input, with some output representation of size 100. In this case $x \in X \subset \mathbb{R}^{m_1 \times m_2}$ has rank 2 with $m_1 = 1600$ and $m_2 = 1000$; $y \in Y \subset \mathbb{R}^{n_1}$ has rank 1 with $n_1 = 100$; and $w \in W \subset \mathbb{R}^{n_1 \times m_1 \times m_2}$ has rank 3. The linear transformation would have a sum over i_1 and i_2:

$$L_w(x)_{j_1} = \sum_{i_1, i_2} w_{j_1 i_1 i_2} x_{i_1 i_2} \tag{3}$$

1.2 Invariance and Equivariance

A function g is said to be invariant with respect to a transformation $T : X \to X$ when g's value is left unchanged:

$$g(x) = g(T(x)) \tag{4}$$

Invariance is supposed to capture notions of a feature being unchanged by some inconsequential alteration to the input. For example, if g is designed to identify rectangles in an image, then a rotation of x by 90° should not affect the outcome of the g. It's still a rectangle after all.

On the other hand, a function g is said to be equivariant with respect to some transformation $T : X \to X$ when $T' \circ g = g \circ T$. T' is chosen based on the properties of the underlying group and the action of that group on input set X and output set Y. In the case that the input space X and Y come from the same underlying space $R^{n_1 \cdots n_M}$, T' is usually just equal to T itself. To simplify matters we will only deal with this specific type of equivariance for the remainder of the paper. One can thus write the equivariance requirement as:

$$T(g(x)) = g(T(x)) \tag{5}$$

Equivariance doesn't require the values of g to be left unchanged by a transformation, but only that the same transformation applied to the ouptut of g gives the same result as applying it first to x.

A common method of capitalizing on these invariant and equivariant properties is to use data augmentation: slight alterations, or more suggestively transformations, to the training examples to account for changes to the data that should not greatly affect changes to representation or corresponding label [7]. The invariance or equivariance is thus learned by seeing many examples from the same class (i.e. related to each other by an invariance transformation). Although effective, this method works by training over extra examples, which can add time to training, and there is still no guarantee the resulting representation will be completely invariant or equivariant to the underlying transformation.

The scope of this paper covers invariance and equivariance with respect to the linear transformation L_w only, rather than a full layer f, although one could certainly extend some of these ideas easily to $A_b \circ L_w$. Extending it to the full f would be more difficult due to σ's typically non affine nature. Moreover, the discussion is limited to linear transformations T. However, one could also extend this reasoning to the non-linear case. Since T is linear, $T(x)$ can be written in tensor notation:

$$T(x)_{i_1 \ldots i_M} = \sum_{i'_1 \ldots i'_M} T_{i_1 \ldots i_M i'_1 \ldots i'_M} x_{i'_1 \ldots i'_M} \tag{6}$$

2 Invariant Transformations

A linear transformation L_w is invariant with respect to T when $L_w(x) = L_w(T(x))\ \forall x \in X$, or in index notation when:

$$\sum_{i_1 \dots i_M} w_{j_1 \dots j_N i_1 \dots i_M} x_{i_1 \dots i_M} = \sum_{\substack{i'_1 \dots i'_M \\ i_1 \dots i_M}} w_{j_1 \dots j_N i'_1 \dots i'_M} T_{i'_1 \dots i'_M i_1 \dots i_M} x_{i_1 \dots i_M} \tag{7}$$

However, by making some rather weak assumptions about the allowed values of $x \in X$ (e.g. there are more than a couple possible allowed values), each term in the sum over $i_1 \dots i_M$ must hold independently:

$$w_{j_1 \dots j_N i_1 \dots i_M} = \sum_{i'_1 \dots i'_M} w_{j_1 \dots j_N i'_1 \dots i'_M} T_{i'_1 \dots i'_M i_1 \dots i_M} \tag{8}$$

otherwise these equation wouldn't hold for each $x \in X$. These equations constrain the possible values of w, so if w is held to satisfy these equations during training, one is guaranteed invariance under T. Therefore, choosing the invariant transformations of the data effectively sets the weight constraint equations of the layer.

2.1 Translation of Indices Invariance

As an example, assume we have some data set where a shift in index by one should not affect its representation. In other words, L_w is invariant with respect to transformations of the form:

$$x_{t i_2 \dots i_M} \to x_{((t+1)\%n_1) i_2 \dots i_M} \tag{9}$$

Where $\%$ denotes modulo and the name t is suggestively given to the index being shifted. The $(t+1)\%n_1$ of (9) ensures the last index 'wraps around' to the first. A transformation T that accomplishes this shift can be written as:

$$T_{t' \dots i'_M t \dots i_M} = [\delta_{t',(t+1)} + \delta_{t',1}\delta_{t,n_1}]\delta_{i'_2,i_2} \dots \delta_{i'_M,i_M} \tag{10}$$

Plugging T into the weight constraint equation gives:

$$w_{j_1 \dots j_N t \dots i_M} = \sum_{t' \dots i'_M} w_{j_1 \dots j_N t' \dots i'_M} [\delta_{t',(t+1)} + \delta_{t',1}\delta_{t,n_1}]\delta_{i'_2,i_2} \dots \delta_{i'_M,i_M} \tag{11}$$

Note that all of the diagonal elements $\delta_{i'_2,i_2} \dots \delta_{i'_M,i_M}$ cancel out the sums, and the i''s are replaced with i's leaving:

$$w_{j_1 \dots j_N t \dots i_M} = \sum_{t'} w_{j_1 \dots j_N t' i_2 \dots i_M} [\delta_{t',(t+1)} + \delta_{t',1}\delta_{t,n_1}] \tag{12}$$

Defining $\bar{w}_t = w_{j_1 \dots j_N t \dots i_M}$ for clarity results in:

$$\bar{w}_t = \sum_{t'} \bar{w}_{t'} \left[\delta_{t',(t+1)} + \delta_{t',1}\delta_{t,n_1}\right] \tag{13}$$

The weight constraint equations thus become:

$$\bar{w}_1 = \bar{w}_2 \quad \bar{w}_2 = \bar{w}_3 \quad \dots \quad \bar{w}_{n_1} = \bar{w}_1 \tag{14}$$

This holds true for every $j_1, j_2, \dots, j_N$ and $i_2, \dots, i_M$, which were swept into the definition of $\bar{w}$. Therefore, the weights must be shared across the whole dimension t. This is, of course, the weight sharing condition of the non-hidden weights (usually denoted U, V, and W) in a recurrent neural network (RNN).

3 Equivariant Transformations

A linear L_w is said to be equivariant with respect to a transformation T when $L_w(T(x)) = T(L_w(x))$. In index notation this can be written as:

$$\sum_{\substack{j'_1 \dots j'_M \\ i_1 \dots i_M}} T_{j_1 \dots j_M j'_1 \dots j'_M} w_{j'_1 \dots j'_M i_1 \dots i_M} x_{i_1 \dots i_M} = \sum_{\substack{i'_1 \dots i'_M \\ i_1 \dots i_M}} w_{j_1 \dots j_M i'_1 \dots i'_M} T_{i'_1 \dots i'_M i_1 \dots i_M} x_{i_1 \dots i_M} \tag{15}$$

Note that the sets X and Y are now assumed to come from the same space $\mathbb{R}^{n_1 \times \cdots \times n_M}$. As in the case of invariance, the equations must hold independently for each term $i_1 \dots i_M$, since the x's are arbitrary. Thus the weight constraint equations become:

$$\sum_{j'_1 \dots j'_M} T_{j_1 \dots j_M j'_1 \dots j'_M} w_{j'_1 \dots j'_M i_1 \dots i_M} = \sum_{i'_1 \dots i'_M} w_{j_1 \dots j_M i'_1 \dots i'_M} T_{i'_1 \dots i'_M i_1 \dots i_M} \tag{16}$$

3.1 Translation of Indices Equivariance

As an example of equivariance we also look at index translation. Similar to what was done in the invariant case, equivariance is imposed with respect to shifts in one of the indices $x_{ii_2 \dots i_M} \to x_{(i+1)i_2 \dots i_M}$. However, unlike the invariant case the ending indices are not 'wrapped around' to the first and this index is simply refered to as i rather than t. At the same time, equivariance is also imposed with respect to shifts in the index's other direction by ω: $x_{ii_2 \dots i_M} \to x_{(i-\omega)i_2 \dots i_M}$. These two transformations can be written as:

$$\begin{aligned} T^1_{i' \dots i'_M i \dots i_M} &= \delta_{i',(i+1)} \delta_{i'_2,i_2} \dots \delta_{i'_M,i_M} \\ T^{-\omega}_{i' \dots i'_M i \dots i_M} &= \delta_{i',(i-\omega)} \delta_{i'_2,i_2} \dots \delta_{i'_M,i_M} \end{aligned} \tag{17}$$

Plugging T^1 into the equivariant weight constraint equation results in:

$$\begin{aligned} \sum_{j' \dots j'_M} \delta_{j,(j'+1)} \delta_{j_2,j'_2} \dots \delta_{j_M,j'_M} w_{j' \dots j'_M i \dots i_M} &= \sum_{i' \dots i'_M} w_{j \dots j_M i'_1 \dots i'_M} \delta_{i',(i+1)} \delta_{i'_2,i_2} \dots \delta_{i'_M,i_M} \\ \sum_{j'} w_{j' j_2 \dots j_M i \dots i_M} \delta_{j,(j'+1)} &= \sum_{i'} w_{j j_2 \dots j_M i' i_2 \dots i_M} \delta_{i',(i+1)} \end{aligned} \tag{18}$$

And then plugging in $T^{-\omega}$ gives:

$$\begin{aligned} \sum_{j' \dots j'_M} \delta_{j,(j'-\omega)} \delta_{j_2,j'_2} \dots \delta_{j_M,j'_M} w_{j' \dots j'_M i \dots i_M} &= \sum_{i' \dots i'_M} w_{j \dots j_M i' \dots i'_M} \delta_{i',(i-\omega)} \delta_{i'_2,i_2} \dots \delta_{i'_M,i_M} \\ \sum_{j'} \delta_{j,(j'-\omega)} w_{j' j_2 \dots j_M i \dots i_M} &= \sum_{i'} w_{j \dots j_M i' i_2 \dots i_M} \delta_{i',(i-\omega)} \end{aligned} \tag{19}$$

For simplicity's sake, we introduce $\bar{w}_{ji} = w_{j \dots j_M i \dots i_M}$. The weight constraint equations become:

$$\begin{aligned} \sum_{j'} \delta_{j,(j'+1)} \bar{w}_{j'i} &= \sum_{i'} \bar{w}_{ji'} \delta_{i',(i+1)} \\ \sum_{j'} \delta_{j,(j'-\omega)} \bar{w}_{j'i} &= \sum_{i'} \bar{w}_{ji'} \delta_{i',(i-\omega)} \end{aligned} \tag{20}$$

The first of these makes constraints of the form:

$$\begin{aligned} &\bar{w}_{11} = \bar{w}_{22}, \quad \bar{w}_{12} = \bar{w}_{23}, \quad \bar{w}_{13} = \bar{w}_{24}, \qquad \dots \\ &\bar{w}_{22} = \bar{w}_{33}, \quad \bar{w}_{23} = \bar{w}_{34}, \qquad \dots \end{aligned} \tag{21}$$

It forces all of the values on the diagonals to be the same. It also forces all the elements below the main diagonal to be zero so that the weights are of the form:

$$\begin{bmatrix} \bar{w}_{11} & \bar{w}_{12} & \bar{w}_{13} & \dots \\ 0 & \bar{w}_{11} & \bar{w}_{12} & \dots \\ 0 & 0 & \bar{w}_{11} & \dots \\ \vdots & \vdots & \vdots & \end{bmatrix} \tag{22}$$

Looking at the $j = 1$ row of the second weight constraint equation, one can find that $\bar{w}_{1k} = 0$ for $k > \omega$. Definining $\bar{w}_k \equiv \bar{w}_{1k}$, the weight constraint equation becomes:

$$\begin{bmatrix} \bar{w}_1 & \bar{w}_2 & \dots & \bar{w}_\omega & 0 & \dots \\ 0 & \bar{w}_1 & \dots & \bar{w}_{(\omega-1)} & \bar{w}_\omega & \dots \\ 0 & 0 & \dots & \bar{w}_{(\omega-2)} & \bar{w}_{(\omega-1)} & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \end{bmatrix} \tag{23}$$

As before, these equations hold for every $j_2, \dots j_M$ and $i_2, \dots i_M$. This is actually the weighting scheme for a convolutional neural network (CNN) layer in 1-dimension with window size ω. Rather than going step by step, multiplying, moving the filter by one and repeating, it is done all as a single matrix multiplication where each row is a interpreted as single step.

If translation equivariance is required for another index, then those transformations can be added as another set of requirements for the weights to fulfill, and which would result in the weight constraints for a 2-dimensional CNN.

It is worth stating that there may be a more natural interpretation of a CNN weights than the described method above. Although, it seems to be successful in producing the desired constraints, a mismatch in step size between translation directions is probably not the first explanation for a CNN's weights that comes to mind. There may be a better interpretation using a different equivariance, a different symmetry type, or perhaps some other completely different method. Further work may need to be done on this topic.

4 Other Examples

This method of extracting weight constraints from invariances and equivariances provides a powerful way to design network layers that automatically adhere to properties of the data, which are known to be there. It appears to be a simple yet effective way to insert insider knowledge into the algorithm itself rather than relying predominantly on training data. Some simple extensions are covered to highlight this.

4.1 Flip of Indices Invariance

A simple yet important type of invariance, especially prominent in image classification, is the mirroring over a particular axis. In most cases the content of an image does not change when the picture is flipped horizontally. An application of this weighting structure can be found in the evaluation section below.

The transformation can be represented mathematically as:

$$T_{i'i'_2\dots i'_M i i_2\dots i_M} = \delta_{i'(i-m_1+1)}\delta_{i'_2 i_2}\dots\delta_{i'_M i_M} \tag{24}$$

with i as the dimension being flipped. The constraint equations become:

$$w_{j_1\dots j_M i\dots i_M} = w_{j_1\dots j_M i'\dots i'_M}\delta_{i'(i-m_1+1)}\delta_{i'_2 i_2}\dots\delta_{i'_M i_M} \tag{25}$$

resulting in the constraints:

$$w_{j_1\dots j_M i\dots i_M} = w_{j_1\dots j_M (i-m_1+1)\dots i_M} \tag{26}$$

Therefore each weight has a partner of equal value which exists on the other side of dimension being flipped.

4.2 Down Scale Invariance

For many image features, the overall scale shouldn't matter. For example, a picture of an eye could be right up against the camera or it could be off in the distance. The scale of the eye has changed but the presence of the eye has not. It may be helpful to capture this invariance in some representation. As a one dimensional example of this invariance, consider:

$$T_{i'i'_2\dots i'_M i i_2\dots i_M} = \frac{1}{n_1}\sum_{s=1}^{S}\delta_{S(i'-1)+s,i}\delta_{i'_2 i_2}\dots\delta_{i'_M i_M} \tag{27}$$

This is interpreted as shrinking the dimension indexed by i by a factor of S. This is accomplished by averaging S values together $\frac{n_1}{S}$ times, leaving the remaining rows as zeros. Plugging this into (8), canceling out the $\delta_{i'i}$'s with the sums, and introducing $\bar{w}_i = w_{j_1 j_2\dots j_N i i_2\dots i_M}$ gives:

$$\bar{w}_i = \sum_{i'}\bar{w}_{i'}\left[\frac{1}{m_1}\sum_{s=1}^{S}\delta_{S(i'-1)+s,i}\right] \tag{28}$$

These relations force constraints of the form:

$$\begin{aligned} \bar{w}_1 &= \bar{w}_2 = \bar{w}_3 = \cdots = \bar{w}_S \\ \bar{w}_{S+1} &= \bar{w}_{S+2} = \cdots = \bar{w}_{2S} \\ &\vdots \end{aligned} \tag{29}$$

i.e. blocks of identical weights:

$$\left[\underbrace{\bar{w}_1 \ldots \bar{w}_1}_{\text{S}} \quad \underbrace{\bar{w}_{S+1} \ldots \bar{w}_{S+1}}_{\text{S}} \cdots \right] \tag{30}$$

4.3 Translation + Down Scale Invariance

These past examples have treated all of the different notions of invariance and equivariance separately, but there is no reason they cannot be combined. If, for instance, one were working on a video classification problem, it might make sense to build a feature which is both time translation invariant and scale invariant. This is the same as saying that both the frame that the feature occurs in and the input's size shouldn't matter.

We assume that the first index is time (t) and the second index (i) is the scale invariant dimension. Defining $\bar{w}_{ti} = w_{j_1 \ldots j_N t i \ldots i_M}$, from (14) and (29) gives:

$$\begin{aligned} \bar{w}_{ti} &= \bar{w}_{(t+1)i} \\ \bar{w}_{t1} &= \bar{w}_{t2} = \cdots = \bar{w}_{tS} \\ \bar{w}_{t(S+1)} &= \bar{w}_{t(S+2)} = \cdots = \bar{w}_{t(2S)} \\ &\vdots \end{aligned} \tag{31}$$

Putting these two sets of constraints together results in weights of the form:

$$\begin{bmatrix} \bar{w}_{11} & \bar{w}_{11} & \cdots & \bar{w}_{1(S+1)} & \bar{w}_{1(S+1)} & \cdots \\ \bar{w}_{11} & \bar{w}_{11} & \cdots & \bar{w}_{1(S+1)} & \bar{w}_{1(S+1)} & \cdots \\ \vdots & \vdots & & \vdots & \vdots & \end{bmatrix} \tag{32}$$

These weights guarantee translational invariance in the t dimension and down scale invariance in the i dimension.

5 Evaluation

As a simple proof of concept that this procedure can used to shrink the number of parameters without drastically affecting accuracy we take the original AlexNet architecture [6] and compare it to a slightly altered version where the first fully connected layer is replaced with a horizontal flip invariant one. The authors of the original AlexNet paper achieved horizontal flip invariance by data augmentation. Every example had a 50% chance of being mirrored about the y-axis, so that the model learned the invariance [6]. This modified AlexNet is instead constrained to exhibit the invariance by the weight sharing scheme. The training and evaluation processes follow the procedure of the original AlexNet paper wherever possible.

5.1 Dataset and Preprocessing

The training and validation datasets from the ImageNet Large Scale Recognition Challenge 2012 (ILSVRC2012) are used in this evaluation [8]. They consist of variable resolution, RGB, human labeled images taken from around the web which fall into exactly one of the 1000 possible label classes. The training dataset is comprised of around 1000 images for each label class resulting in roughly 1.2 million images in total. The validation dataset has 50,000 labeled images. The ILSVRC2012 has a test set but it is not used here [8].

The same preprocessing used in the AlexNet paper is also used here. The variable resolution images are all scaled down to a fixed size of 256×256 by taking the minimum of the height and the width

and scaling the picture isotropically so that the minimum side is 256 pixels long. Then, the longer of the two sides is center cropped so that it too had a length of 256, resulting in a picture of size 256×256. Afterward, the average RGB pixel value taken from the training set is subtracted from every pixel of every image, leaving a zero RGB mean dataset [6].

5.2 Benchmark AlexNet

To study the effects of adding in a flip invariant layer to a neural network, a model which can be simply altered is needed. The architecture and training process described in the original AlexNet paper serves as a manageable and easily recognizable benchmark. A relatively brief description of the process is given here, but it follows very closely to that of the original paper. For more detailed information refer to the original paper [6].

Before feeding the preprocessed images into the model, there are two distinct forms of data augmentation applied: a random horizontal flip and a random 224×224 crop from the 256×256 preprocessed image. The RGB value skewing described in the original paper is ommitted for simplicity. As remarked above, the random flip allows for a horizontal flip invariance to be learned by randomly picking the left to right orientation of the image. The random crop allows for small translation perturbations. At test time, the softmax values from the four corner crops and center crop as well as their corresponding horizontally flipped version are averaged together to get the final prediction.

The overall architecture of the model is the 5 CNNs model referred to in [6]. It is comprised of five convolutional layers, two fully connected layers and a softmax layer in that order. The first two convolutional layers each contain local response normalization and max pooling layers while the fifth layer contains only the max pooling. The third and fourth layers contain neither local response normalization nor max pooling. Dropout was used as a regularization technique on both of the fully connected layers. Each convolutional and fully connect layers use relu activations. The sizes of the layers are given in the AlexNet paper [6]. The model definition and training process is done in tensorflow [1] based off of an implementation taken from [2].

For the training process, a weigth decay term is included. The L2 norm of all trainable weights (excluding biases) multiplied by a weight decay factor of 0.0005 is added to the cross entropy to get the total loss. The model is then trained using stochastic gradient descent with momentum equal to 0.9. The learning rate is first set to 0.01, trained until the validation accuracy holds relatively steady and divided by a factor of 10. This is repeated several times for learning rates 0.01, 0.001, 0.0001 and 0.00001. A batch size of 128 is used [6].

5.3 Flip Invariant AlexNet

Both the architecture and training process of the flip invariant version of the AlexNet (FIAN) are nearly identical to that of the benchmark AlexNet (BAN). The architecture of FIAN differs from BAN only in the sixth layer. BAN's sixth layer, a fully connected one, is replaced with a flip invariant one in FIAN.

The output from the fifth convolutional layer is of shape $(6, 6, 256)$, i.e. height six, width six and 256 channels. In the BAN, that output is flattened to a vector of size $6 \times 6 \times 256 = 9216$ and fed to the fully connected layer, which outputs to a vector of size 4096. The FIAN keeps the height and width dimesions, feeds it through a layer that is invariant under flips in the width dimensions of shape $(6, 6, 113)$. The 113 channels are chosen in order to roughly match the 4096 of the BAN since $6 \times 113 = 4068$. There are of course actually only $4068/2 = 2034$ free parameters, since this is a flip invariant layer. The rest of the FIAN follows along the BAN structure. The 6×113 output from the flip invariant layer are flattened to a vector, fed through a fully connected layer and then a softmax layer.

The training process of FIAN also looks very similar to that of BAN. The only main difference is the value of the weight decay factor. Due to the fact that there are roughly half the number of free parameters in the sixth layer, the weight decay factor need to be increased to 0.001. This value is chosen in an empircal manner such that the total weight penalty contribution to the loss of FIAN match roughly that of BAN.

5.4 Results

With such a reduction in the number of free parameters, a large drop in the accuracy would be expected. However, this is not the case. The BAN's top-1 validation error of 42.63% is only 0.23% less than that of the FIAN at 42.86%. The top-5 validation error on the other hand is more comparable and the FIAN actually performs marginally better: 20.07% for the BAN and 19.88% for the FIAN.

Where the two models differ the most is in the training process itself. The BAN finished it training in about 75 epochs, while the FIAN's reduced number of parameters allows for a lower number of training steps of about 55 epochs. The training times of the two are 80 hours and 61 hours for the BAN and FIAN respectively. They were trained using a GeForce GTX 1060 GPU with 6GB of memory. The increase in per step training time for the FIAN is most likely caused by the memory consumption. Despite the fewer number of parameters, the 6th layer of the FIAN consumes substianlly more memory (500MB) than that of the BAN (8MB). This is due to the fact that there currently is no tensorflow operation that creates a flip invariant layer, so several operations such as transpose, reverse, and concatenation have to be used in order to constrain the weights.

6 Conclusions

Under the proposed method, the design of a DNN would look a little different than the status quo. One would start with the input data and attempt to understand what the lower level representations are, e.g. corners, straight lines, verb, etc. Then the transformations that the representation should be invariant or equivariant under would be determined, and the representation's weigths would be constrained to respect that invariance or equivariance. Higher level representations (e.g dogs, sentence clauses, landscapes) can be done in much the same way. Then the model can be trained as any DNN. With the reduced number of parameters the networks could be larger and more complicated, and would hold all of the properties of a model normally achieved through data augmentation or other methods.

There are a few things worth noting. First, all of the transformations presented in this paper come from finite groups. Any attempt at constraining weights to respect some continuous invariance or equivariance seems to result in the weights being over constrained. Perhaps not terribly surprising, but worth acknowledging nevertheless.

As mentioned before, the weight constraints are all made to statisify purely linear representations. When putting linear representations through the translation A_b and activation σ, the invariances and equivariances are very unlikely to hold. The success of the CNN's and RNN's in recent years has shown that there are still benefits to encoding these relationships into the network even if they aren't preserved completely. Furthermore, it should be possible to find weight constraints which guarantee invariance or equivariance under functions more general than linear ones. A neat, closed-form solution may not exist, but many such weight constraint equations should be able to be solved using numerical methods.

Thus far all of the weight constraints have been of the form $w_i = w_j$, but it might be beneficial to be able to make softer constraints. Very often the data may be invariant under some transformation up to a point. For example, a feature could be the same so long as it is not translated too far. However, the method above implies an invariance for all members of the group, in this case all translations. It would be helpful if there was some way to encode a softer relation between w_1 and w_2, which would allow for less stringent restrictions.

The proposed method allows for the discovery of many new network designs. The evaluation showed that it is possible, at the very least, to reduce the size of a model and speed up training. Only a select few transformations T are covered in this paper, and in most cases their simplest forms. With some work, this method seems to have the potential to be a helpful tool in neural network design.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore,

Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Michael Guerzhoy and Davi Frossard. Alexnet implementation in tensorflow, with weights. `http://www.cs.toronto.edu/~guerzhoy/tf_alexnet/`, 2016.

[3] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[4] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. Large-scale video classification with convolutional neural networks. In *CVPR*, 2014.

[5] Alex Krizhevsky, I Sutskever, and G. E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems (NIPS 2012)*, page 4, 2012.

[6] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[7] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

[8] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[9] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[10] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, January 2014.

[11] Oriol Vinyals, Ł ukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey Hinton. Grammar as a foreign language. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2773–2781. Curran Associates, Inc., 2015.

[12] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. *CoRR*, abs/1707.07012, 2017.