Ben-Gurion University of the Negev

The Faculty of Natural Sciences

The Department of Computer Science

The Effect of Alignment and Regularization on STN-based Classification

Thesis submitted in partial fulfillment of the

requirements for the Master of Sciences degree

Asher Fredman

Under the supervision of:

Dr. Oren Freifeld

September 2019

Ben-Gurion University
of the Negev

Ben-Gurion University of the Negev

The Faculty of Natural Sciences

The Department of Computer Science

The Effect of Alignment and Regularization on STN-based Classification

Thesis submitted in partial fulfillment of the

requirements for the Master of Sciences degree

Asher Fredman

Under the supervision of:

Dr. Oren Freifeld

Signature of student: _____ Date: 11.09.2019

Signature of supervisor: Oren Freifeld Date: 11.09.2019

Signature of chairperson of the committee for graduate studies: _____ Date: 24.10.2019

September 2019

Ben-Gurion University of the Negev

Ben-Gurion University of the Negev

Thesis submitted in partial fulfillment of the
requirements for the Master of Sciences degree

Asher Fredman

September 2019

The Effect of Alignment and Regularization on STN-based Classification

Abstract

Convolutional Neural Networks (CNNs) have led to drastic performance gains in classification tasks. One drawback of CNNs, however, is their limited invariance to most spatial transformations. One popular approach that tries alleviating this problem is using a Spatial Transformer Network (STN); namely, a network that learns and applies spatial transformations to its input such that it should help decreasing the learning loss. Unfortunately, however, practice shows that it is often hard to train STNs successfully, especially when STNs are incorporated in very deep nets such as DenseNet. With the goal of improving the training of STN-based classifiers, we study two modifications to the classification loss: a regularization term and an alignment term. The oft-used regularization term has not been well studied. Our work demonstrates that, in some cases, not using a regularization term during an STN-based classification process results in bad transformations which harm the overall classification objective. The proposed alignment term is new in the context of improving classifier training. The main empirical conclusion of our work is that when one may expect alignment to be feasible, using the auxiliary alignment term boosts performances, even in very deep networks. We demonstrate the positive effect of the alignment term on the Cluttered MNIST dataset (using a simple network) and on Google's SVHN dataset (using either a simple network or DenseNet).

Acknowledgments

I am extremely fortunate to have performed this research under the supervision of Dr. Oren Freifeld. I thank Dr. Freifeld for the knowledge that he bestowed, for his close guidance and for his uncompromising commitment to the quality of the research.

This research was supported, in part, by the Helmsley Charitable Trust through the Agricultural, Biological and Cognitive Robotics Initiative and by the Marcus Endowment Fund both at Ben-Gurion University of the Negev.

I would also like to thank my parents, Joel and Martha Fredman, for always believing in me and for their constant support in everything I do.

I will end by thanking my dear wife Elisheva, and my wonderful children, for being a loving and supportive family, and providing me with the strength and happiness needed to accomplish and succeed despite life's various challenges. I could not have accomplished this work without them.

Contents

1 Introduction **1**

1.1 General Description 1

1.2 Challenges 4

2 Related Work **7**

2.1 Artificial Neural Networks 7

2.1.1 Artificial Neurons 7

2.1.2 Fully Connected Networks 8

2.1.3 Training Networks 10

2.1.4 Stochastic Gradient Descent and Backpropagation 11

2.2 Convolutional Neural Networks 15

2.2.1 Convolutions 15

2.2.2 Convolutional Neural Networks 17

2.2.3 Feature Maps and Feature Extraction 20

2.3 Very Deep Networks 22

2.3.1 Introduction to Very Deep Nets 22

2.3.2 ResNet 23

2.3.3 DenseNet 25

2.4 Image Warping 26

2.4.1 Spatial Transformations 27

2.4.2 Affine Transformations 27

2.4.3 Backward Warping 29

2.4.4 Bilinear Interpolation 32

2.5 Spatial Transformer Networks 34

2.5.1 STN Introduction 34

2.5.2 STN Architecture . . . 35
2.5.3 STN and Backprop . . . 36
2.6 Joint Alignment . . . 37
2.7 Motivation for the Proposed Method . . . 38

3 The Proposed Method **42**
3.1 Method . . . 42
3.1.1 The Alignment Loss Term . . . 42
3.1.2 Optional Regularization . . . 46
3.1.3 Training . . . 48

4 Results **50**
4.1 Results Using The Method . . . 50
4.1.1 Datasets . . . 50
4.1.2 Baseline Architectures . . . 51
4.1.3 Number of ST Layers . . . 51
4.1.4 General Experiment Design and Analysis . . . 51
4.1.5 Cluttered MNIST: 40-by-40 . . . 52
4.1.6 Cluttered MNIST: 60-by-60 . . . 52
4.1.7 SVHN . . . 53
4.1.8 Limitations of the Proposed Method. . . . 54

5 Conclusion **55**

Appendix A **56**
A.1 Spatial Transformer (ST) Layers . . . 56
A.1.1 Localization Network . . . 56
A.1.2 Optimization . . . 56
A.2 Cluttered MNIST . . . 57
A.2.1 Conv Layers . . . 57
A.2.2 FC Layers . . . 57
A.3 SVHN with a “Shallow” CNN as Baseline . . . 57
A.3.1 Conv Layers . . . 58
A.3.2 FC Layers . . . 58

A.4 SVHN with DenseNet as Baseline 58

List of Figures

1.1 Visualizing the disadvantages of a regular CNN 2
1.2 40-by-40 Cluttered MNIST alignment 3

2.1 The artificial neuron 9
2.2 Backpropagation simple example 13
2.3 Convolution 16
2.4 CNN architecture and feature maps 18
2.5 Features generated by CNNs 21
2.6 CNN depths in the ILSVRC competition over the years 23
2.7 Sigmoid and its derivative 24
2.8 Resnet's skip connection 25
2.9 The DenseNet architecture 26
2.10 Affine transformations 30
2.11 Forward warping Vs. Backward warping 31
2.12 Bilinear Interpolation 34
2.13 STN architecture 35
2.14 Congealing 39
2.15 60-by-60 Cluttered MNIST alignment 40

3.1 Proposed architecture 43
3.2 Box plot and error rates for 60-by-60 Cluttered MNIST 45
3.3 Box plot and error rates for SVHN with simple CNN 46
3.4 Box plot and error rates for reduced SVHN with DenseNet 47
3.5 The importance of STN regularization 48
3.6 Limitations of the proposed method 49

4.1 Datasets 51

Chapter 1

Introduction

1.1 General Description

Artificial Neural Networks (ANNs) and specifically Convolutional Neural Networks (CNNs) have led to drastic performance gains in different fields in Computer Vision, such as classification [23, 46, 48, 13, 17, 16], detection [10, 39, 27, 42, 43, 51] and segmentation [41, 32, 35, 1, 4]. This is generally done by learning good features which represent the images, and learning how to correctly classify those features.

One drawback of CNNs, however, is their limited invariance to most spatial transformations. While there are CNNs that were designed to handle simple transformations as translation, invariance to more complex transformations is still an open challenge [14, 52]. Take for example a visual-classification task on a dataset constructed of handwritten digits such as the MNIST dataset [25]. In this dataset there are 50,000 training images which are size-normalized and centered in a fixed-size 28×28 grid. A simple CNN classifier trained on MNIST can easily obtain close to 100% classification accuracy on the test data, for which all images are also centered similarly to the training data. Since the classifier was trained on many images which are located more or less in the same location with minor translation differences, then even if we create a new test image which is translated by a small amount to any direction, the classifier would still be able to correctly classify the digit. However, if the digit was to be translated by a more substantial amount, and more so if we were to perform different transformations such as rotation and/or scaling, then completely wrong classifications would be generated (see Fig. 1.1).

Two leading paradigms for trying to alleviate this problem are data augmentation

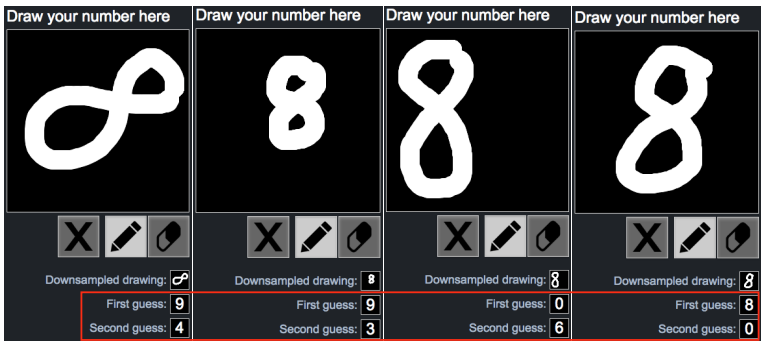


Figure 1.1: Visualizing the disadvantages of a regular CNN. We use here the CNN visualization tool created by Harley [11]. The CNN has two convolution layers followed by two fully-connected layers, and was trained on the MNIST dataset. We see that when the test image is positioned similarly to the training images, good classification is achieved (rightmost image). On the other hand, when we perform different transformations (*e.g.*, translation, scale and rotation), the classifier is completely wrong.

(*e.g.*, [44, 45, 12, 33]) and using a Spatial Transformer Network (STN) [21, 26, 47, 49]). In the first technique, many transformed versions of the original training images are added to the training data. This requires a separate preprocessing stage, and of course can substantially extend the required learning time due to the extra data. This thesis focuses on the second technique, which was first proposed by Jaderberg *et al.* [21]. An STN is a network which has a modular nature such that it can be incorporated in other networks, regardless of the task at hand. Therefore, we also refer to STNs as Spatial Transformer (ST) layers; namely, layers that learn and apply spatial transformations to their input such that they should help decrease the overall learning loss [2, 30]. STNs are discussed in further detail in section 2.5.

Unfortunately, and despite their elegance, training STNs for real-world image classification tasks often turns out to be difficult in practice. In fact, often STNs yield worse results than the results achieved by the baseline architecture without an STN (see section 4.1 for a typical example). This is especially true when the baseline is a very deep network such as DenseNet [17]. Very deep networks, as well as the difficulties of incorporating ST layers in them, are further discussed in section 2.7 below.

As previously mentioned, an ST layer applies spatial transformations to its input such that it should help decreasing the learning loss at hand. These learned spatial transformations do not necessarily cause the input to become aligned. To facilitate better

Figure 1.2: Aligning Cluttered MNIST. Rows 1-3 are random images from the Cluttered MNIST dataset (here shown the 40-by-40 case; see Fig. 4.1 for the harder case). Row 4 is the average of all test images per class, blurred due to misalignment. Row 5 is the average of the transformed test images using an STN trained with the proposed alignment term. The sharpening of the result attest to the success of the alignment.

training of STN-based classifiers, we propose modifying the classification loss via the addition of an alignment term (see Fig. 1.2). This addition is novel in the context of training STN-based classifiers. By adding the auxiliary alignment term, the alignment is learned during the classification learning process, giving an end-to-end learning approach. We have also studied the effect of the addition of another term which regularizes the transformations. While the regularization term is often used in STNs, the effects of its usage were little studied and usually are not reported. In contrast, in our work we have performed a thorough empirical analysis of the usage of these two terms (both together and each one separately).

The empirical conclusion suggests that in datasets for which alignment makes sense, one should always use the alignment term. In other words, our results indicate that an STN with an alignment term boosts STN performance in all cases (regardless of whether, in addition to that term, the regularization term is used as well). More specifically, we show that using the auxiliary alignment loss brings an improvement over the baseline (no STN), over an STN with the original loss, and over using an STN with regularization but without the alignment term. This useful finding holds true regardless of whether the

traditional usage of the STN (*i.e.*, without the alignment term and either with or without regularization) improves over the baseline or not. The question of whether, in addition to using the alignment loss, one should also use the regularization loss depends on the complexity of the dataset and/or baseline architectures. Our work demonstrates that there is a positive effect of the alignment term on the Cluttered MNIST dataset (using a simple network) and on Google's SVHN dataset (using either a simple network or DenseNet). In the former case, we find that it is better to use the alignment term without regularization, while in the latter case it is better to use both. We also show a negative example, on a dataset (CIFAR) where alignment (using homographies) is impractical; in such case, the alignment term did not improve classification performance.

Summary of our contributions. To the best of our knowledge, this is the first work which emphasizes, analyzes, and demonstrates the effect of using STN regularization. We have strived to do so in a thorough manner via multiple runs in different settings, with the results summarized in informative boxplots. Moreover, *the main contribution of this work is the proposed auxiliary alignment loss and in showing (through thorough experiments and analysis), that using it is the preferred strategy for training STN-based classifiers*. In other words, in datasets where alignment makes sense, pushing the STN to learn transformations that not only help decreasing the classification loss but also align certain feature maps to decrease an explicit alignment loss, improves classification.

1.2 Challenges

Our main research goal was to examine whether aligning the images as part of the learning process of a classification task can help achieve better accuracy. We quickly faced a few challenges:

- **STN Regularization**

  We noticed that for certain datasets (see in section 4.1 for more detailed reference) it is hard to train STNs. We found that the networks achieve better results without the addition of ST-layers, and that adding the ST-layers harms the classification process. This is especially true for very deep networks for which using an STN without any modifications usually ruins the learning process and yields classification accuracies similar to those obtained by random guessing.

Therefore we decided to regularize the STN's learned parameters. This endowed the STN with the needed stability and at times allowed it to outperform the baseline network.

Though we scanned much of the STN literature in search of a reference addressing or analyzing the need for such a regularization term, most of the scanned literature did not mention it, and that which did mention that some form of regularization had been used neither provided details nor analyzed the importance of STN regularization. Therefore, we ran multiple experiments on different datasets and different networks, and have demonstrated this importance through informative boxplots.

- **Joint Alignment in CNNs**

  In our work we performed joint image alignment, where the alignment is measured using the popular least-squares criterion (this choice is discussed in section 2.6). The challenge we faced was choosing the correct location in the network for measuring this alignment, namely, which feature maps should be used for the joint alignment task. The most obvious choice was to use the input images themselves as the feature maps to be aligned. This way we would hopefully transform the images themselves and align them immediately before moving on to the network which would perform the classification.

  The problem that we grappled with is that it is not possible to align two different RGB images, which belong to the same class, based on the variance between the pixel values at each pixel location. This is because of the differences naturally found between images even if they belong to the same class. These differences include different backgrounds, lighting, color *etc*.

  With this in mind, we chose to align the feature maps generated as part of the CNN learning process, since these would generally be composed of more descriptive features in which the external noise is usually not visible.

- **Alignment with STNs in very deep networks**

  Another challenge grappled with was that at first glance it seemed that even with the added regularization, very deep networks such as DenseNet do not gain by the added functionally obtained by adding ST-layers into the network. As described

in section 2.7 below, this is due to the multiple down sampling stages and to the ability to learn richer and more descriptive features by going deeper into the network. In these very deep networks we could get good classification even without the ST layers, and so the need to learn the extra transformation during the learning process might harm or not improve the results. Therefore, we set out to examine whether such state-of-the-art networks can gain from spatial invariance and, more specifically, from alignment functionality.

Chapter 2

Related Work

In this chapter we explore the mathematical and technical details relevant to our work.

2.1 Artificial Neural Networks

The emergence of Artificial Neural Networks (ANNs) and especially Convolutional Neural Networks (CNNs) has drastically changed the field of computer vision, and strongly influenced the common approaches to classification tasks [15, 23, 54]. An extensive literature survey of CNNs is beyond the scope of this work, and can be found in *e.g.* [37, 54]. Here we present only a brief overview of the methods most relevant to the proposed approach. The networks discussed below perform classification similar to the task we performed in our work, bearing in mind (as noted previously) that CNNs can get state-of-the-art results in many other tasks as well.

2.1.1 Artificial Neurons

The most basic module constructing an Artificial Neural Network (ANN) is an Artificial Neuron (which is further referred to herein as simply a neuron). A neuron is a function which accepts an input x, performs on it a linear operation of the form

$$z = w^\top x + b \tag{2.1}$$

(which is linear with respect to w and b), then performs on the output z a nonlinear function (known as an activation function), and finally returns the activation function's output.

Popular choices for activation functions are sigmoid:

$$\sigma(z) = \frac{1}{1 + e^{-z}}, \tag{2.2}$$

tanh:

$$\sigma(z) = \frac{2}{1 + e^{-2z}}, \tag{2.3}$$

and a Rectified Linear Unit (ReLU):

$$\sigma(z) = \max\{0, z\}. \tag{2.4}$$

In practice, the sigmoid activation function is less in use these days (besides as a loss function) due to certain limitations it has which affect the learning process (see section 2.3.2 for more details). We will just point out that we can append the bias value b to the end of w, forming a new vector $w' = [w, b]$ and update the input vector x by adding to it a homogeneous coordinate, forming $x' = [x, 1]$. Now by taking the dot product of x' and w' we arrive at the previous z: $x'^{\top} w' = w^{\top} x + b = z$. We will work in this setting since in deep learning we perform many matrix multiplications, and many coding languages work faster in vectorized forms.

We will shortly describe how deep ANNs are created by stacking neurons into a layer and connecting those neurons to other neurons in the proceeding layer. This means that each neuron in any given layer takes a weighted sum of values sent to it from the previous layer, and outputs a certain value to the next layer. This behavior, of weighing the inputs and forwarding the output of each neuron to the following neighboring neurons, slightly resembles the behavior of the real biological neuron, which receives inputs from its neighboring neurons, and as a result of those inputs the neuron is capable of reaching some threshold which will activate the neuron and cause it to "fire" an electrochemical message to its neighbors (see Fig. 2.1).

2.1.2 Fully Connected Networks

A Fully Connected (FC) network is generally constructed of a number of layers. The first layer is the vectorized version of the input image which we would like to classify. After the input layer we will have a varying number of layers where each layer is constructed of neurons. The last layer in the network is the output layer which is a loss value indicating

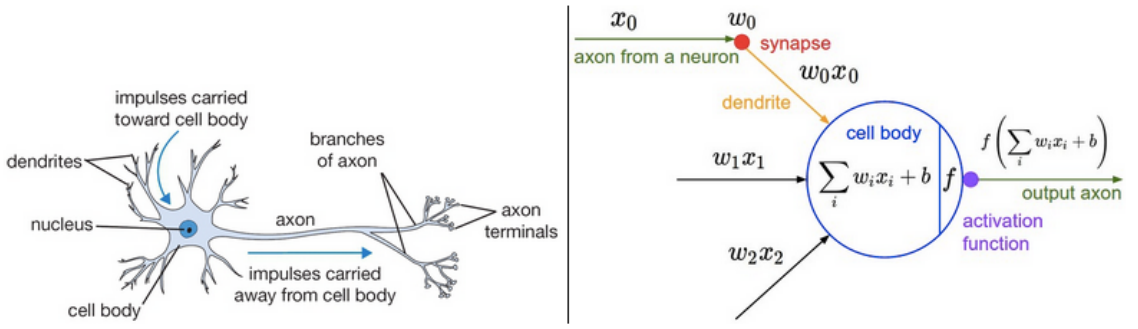


Figure 2.1: From biological neurons to artificial neurons. Image Credit: cs231n.github.io

how well the network classified the image. A "hidden layer" is a layer constructed of neurons (*i.e.*, not the input layer) and is not the output layer. A "deep" Neural Network is a network with at least one hidden layer. When each node in a certain layer is connected to each node is the following layer, we say that the layers are FC or Dense. A FC network is a network where all its layers are FC.

As opposed to traditional machine-learning (ML) techniques which require selecting good representative features to use as input to a ML model, in deep learning the network is required to create a good set of features which will help the classification objective. These features are essentially just a layer of neurons, and so the network needs to learn the best parameters (the weights used to calculate each neuron) in order to generate such features.

Loss functions are functions which indicate how well the network performed on the classification task. For the case of binary-class classification, we have two possible classes: 0 or 1. In this case the loss function which is usually chosen is based on the sigmoid function mentioned previously, which outputs $\sigma(z) = \frac{1}{1+e^{-z}}$ for which the output value belongs to the interval $(0, 1)$. Since there are two possible classes which the networks can classify the images by, the sigmoid function, which gives each class a probability, is often employed as a mechanism to decide which is the correct class of the input image. The loss function which is chosen is the Logistic Regression function which pushes the learned probability to be as close as possible to the correct class. The loss value for the i-th image is thus:

$$L_i = -y_i \log(\sigma(z_i)) - (1 - y_i)\log(1 - \sigma(z_i)). \tag{2.5}$$

In the case of multi-class classification there are C possible classes and the correct label for each image is represented by a one-hot vector (*i.e.*, a vector with C entries, with 1 in the entry of the correct class and 0 everywhere else). The Softmax function is used

in order to generate a probability over the C classes. In this method, the last layer of the network is a FC layer composed of C neurons for which we calculate the linear function (but not the nonlinear function we mentioned), which produces a score vector with C entries. We then take the exponent of this score vector and normalize the values to get a probability for each of the classes. Finally we take the dot product between the correct one-hot label vector y and the minus log of the softmax vector. This is also known as the Cross-Entropy loss. For example if the final layer produces a score vector s, then the loss value for the i-th image is:

$$L_i = -y^\top \log\left(\frac{e^s}{\sum_{c\in C} e^{s[c]}}\right). \tag{2.6}$$

where e^s is the element wise exponential operation of the entries in s, and $s[\bullet]$ is the value of s at a certain entry. Finally the average over the loss for each image in the batch is returned. As noted, the softmax loss function can be viewed as a probability distribution. Namely, for a given image x_i the network gives each possible class $c \in C$ a probability that x_i belongs to that class, where $P(x_i \in c) \geq 0$ and $\sum_{c\in C} P(x_i \in c) = 1$. This is a probabilistic interpretation for the softmax function. As an aside it is noted that there is another information-theoretic view of the softmax function which connects the Cross-Entropy loss function with the Logistic loss function.

2.1.3 Training Networks

In the visual classification task in machine learning and specifically in deep learning, the goal is to create a model which can correctly classify a previously unseen set of images. The model in deep learning is the neural network, and is usually obtained by performing what is called 'supervised learning'. In supervised learning we are given an input (X, y) where X is set of n points (*e.g.*, images) $x_1, x_2, ..., x_n \in \mathbb{R}^m$, and $y \in \mathbb{R}^n$ is a vector of corresponding n labels. The process which generates an image classifier is broken down into two stages, the training stage and the test stage, in which the input data is broken down into three different groups: training data, validation data and test data. In the first stage, the training stage, we perform many iterations on the training data where in each iteration we update the values of the network's parameters in a way which minimizes the overall loss function obtained on the training data. After each iteration we can validate that the learning is proceeding as expected (and that we are not 'overfitting' the model to be

perfect only for the training data and not for other random points) by testing the results on the validation set. If needed we can change different aspects of the network and begin the training stage again. The training will continue until a certain stopping criterion is met (such as a certain number of iterations or until we reach the point where the network begins to overfit). After the training is finished there is the test stage, in which the model is tested on the test data. The test data is assumed to be random points generated from the same probability distribution of the training points, and so if the model achieves a good classification score on the test data then the neural network classifier has been successfully trained for classifying images generated by that distribution.

2.1.4 Stochastic Gradient Descent and Backpropagation

As already mentioned, each neuron can be viewed as a function on an input vector x_i and be defined via a weight vector w. Since we have many neurons in a given network, the entire network can be viewed as a very large function $f(x_i; \theta)$ which given a group of parameters θ, computes a classification score vector for the input point x_i. Following that logic, the loss function $L(x_i, y_i; \theta)$ which uses the produced score vector in order to calculate the loss, is also one big function which uses the outputs of $f(x_i; \theta)$ and the correct label y_i in order to produce a loss score. The task is to find the best set of parameters θ so that the function f would calculate the best score vector possible and the function L would return a minimal value.

A loss function which is based on an ANN might very likely be composed of millions of parameters, since each layer of neurons in the network has its own weight matrix which defines the neurons in that layer and where this matrix might be quite large (see following section where we discuss the reasons for using CNNs). The question is how do we train a network with millions of different parameters so that we would minimize the loss? The answer is the Stochastic Gradient Descent (SGD) algorithm [22, 5]. SGD is a well-known algorithm when it comes to optimizing functions, and different variants of SGD are widely used in deep learning. SGD itself is a variant of another famous algorithm called Gradient Descent (SG), which is an iterative algorithm for learning an optimal value for a set of parameters θ so that the overall function $f(x; \theta)$ gets minimized (or maximized) for a given input X, y where X is generally a set of images needed to be classified and y is the corresponding labels used to calculate the loss function. At each iteration of the

GD algorithm, we calculate the loss function for the entire data, and then calculate the gradient of all the parameters constructing the loss function. After that we update the values of the parameters by taking a step in the opposite direction of the gradient (since the gradient points to the direction needed in order to maximize the function, and here we are minimizing it). A good analogy for the GD algorithm is that of a blindfolded hiker who is placed on a hilltop and needs to make his way down to the valley. A safe way for him to do so is to feel which direction is that of the steepest ascent, and take a safe step (possibly not too big since he doesn't know where he will end up, but not to small either if he would like to make it home in time for dinner) in the opposite direction.

The regular GD algorithm is very time consuming mainly due to calculating the gradient on the entire training data. The SGD (or 'Mini-Batch GD') works the same way as the GD algorithm only that in each iteration instead of calculating the loss function for all the points in the training data, we take a random subset of points and calculate the loss and gradients for it. This subset of random points is called a mini batch and the subset's size is referred to as the batch size. The extreme case is to use a batch size of one (which is the case in SGD). The idea is that even if we were to update the parameters at each iteration based on the loss function generated by a single random point, the gradient of the expectation of an infinite amount of such iterations would converge to the correct gradient leading us to the minimum of the entire function. Let $\alpha^{(k)}$ and $\theta^{(k)}$ be the step size and parameters computed at iteration k. The algorithm works as follows: While a stopping criterion is not met:

1. Sample n training points X and their correct labels y.

2. Compute the average loss of the points in the batch $L(X, y; \theta^{(k-1)})$.

3. Compute the gradients $\hat{g}$ of $L(X, y; \theta^{(k-1)})$ given the loss value.

4. Update $\theta^k = \theta^{(k-1)} - \alpha^{(k)} \times \hat{g}$.

The question is how to calculate the gradient of the batch in an efficient way. We could calculate the gradient numerically which is quite easy to compute: Suppose $|\theta| = u$. In the numerical approach we iterate over all the parameters and in each iteration increase the value of a different parameter θ_i by a very small step h. The partial derivative of the loss

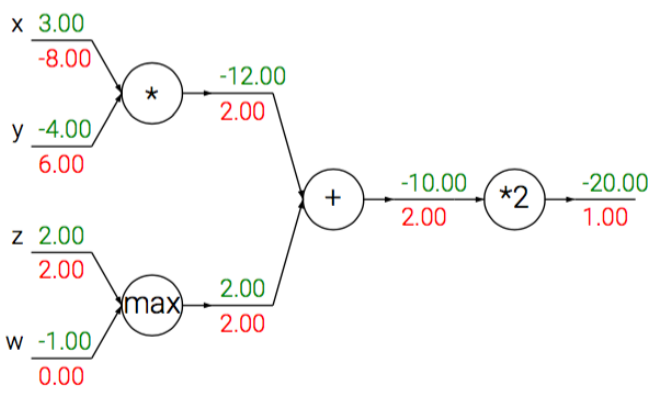


Figure 2.2: Parameter update using the chain rule for the function $L(x, y, z, w) = (x \times y + max(z, w))^2$. Given initial values for the different parameters we first calculate the function's value via a forward pass (values in green), and then we find the partial derivative of each parameter with respect to the overall function. This is done by applying the chain rule (values in red). Credit: cs231n.github.io

function L with respect to θ_i is approximately:

$$\frac{\partial L(\theta_1, ..., \theta_i, ...\theta_u)}{\partial \theta_i} \approx \frac{L(\theta_1, ..., \theta_i + h, ...\theta_u) - L(\theta_1, ..., \theta_i, ...\theta_u)}{h}. \tag{2.7}$$

There are two disadvantages to this approach. The first is that this approach requires updating each parameter separately for which the complexity is linear in the amount of parameters, and as mentioned we may have millions of parameters. Another disadvantage is that this approach is a close approximation but not an exact solution. Because of this we calculate an analytic solution, which although the calculation itself is more complex and thus more error prone than the simple numerical approach, it produces an exact solution in a much more time efficient manner. In order to calculate the gradient analytically we start at the final layer outputting the loss and calculate the way back to each of the parameters while using the chain rule.

Backpropagation [36, 3] is an algorithm used at the heart of neural network training which simply applies the chain-rule in order to learn the parameter update as part of the SGD algorithm. Training neural networks is broken down into iterations and epochs. An iteration is composed of a single forward pass and backward pass (referred to as backpropagation) of an entire mini batch. During an iteration we iterate over all the data in the mini batch and for each point run a forward pass through the network which will output a loss value for that point. After running all the points in the batch through the network,

we take the average loss value and use it in order to update the network's parameters via backpropagation. Once we have finished with one mini batch we move on to the next mini batch in the data and run an iteration on it. Once we have finished going over the entire data, we have performed one epoch. It is common to perform quite a few of such epochs during training. The value of the learning rate parameter α is usually decreased every few epochs. The idea here is that after the network took a few relatively large steps in a good direction, we are nearing in on a good local minimum in the loss function's landscape, and therefore we would like to take smaller and smaller steps, so as not to step over the minimum point.

Single iteration in a Fully connected (FC) network. We will now examine an L-layer FC network, where the images it takes as input have n pixels and where we stack each of the image's pixels into a R^n vector before feeding it into the network. The task of the network is to correctly classify each of the images given k different classes. This implies that the last layer of the network is a FC layer which would have k neurons representing the scores of the k classes. Using the output of the final layer we will run the softmax function to produce loss value for the ith image while using the correct label y_i. Let us define the vector of neurons in layer $l+1$ to be

$$x_i^{(l+1)} = h_l(W^{(l)}, x_i^{(l)}) \tag{2.8}$$

for layers $l = 1, ..L-1$, where $x_i^{(0)}$ is the ith input image and h_l is an activation function used in order to produce the neurons in layer $l+1$. h_l could be any of the nonlinear activation function we have previously seen (*e.g.*, relu) or it can be a linear identity function (which is what is used to produce the final layer) *i.e.*

$$x_i^{(L)} = h_{L-1}(W^{(L-1)}, x_i^{(L-1)}) = W^{(L-1)} x_i^{(L-1)}. \tag{2.9}$$

Now the network's loss function for a single image is simply

$$L_i(x_i^{(L)}, y_i), \tag{2.10}$$

where L_i is the loss value for image i. Assume now that our network has 3 layers, in which case our entire loss function can be represented as:

$$L_i(x_i^{(3)}, y_i) = L_i(h_2(W^{(2)}, x_i^{(2)}), y_i). \tag{2.11}$$

$$L_i(h_2(W^{(2)}, x_i^{(2)}), y_i) = L_i(h_2(W^{(2)}, h_1(W^{(1)}, x_i^{(1)})), y_i). \tag{2.12}$$

$$L_i(h_2(W^{(2)}, h_1(W^{(1)}, x_i^{(1)})), y_i) = L_i(h_2(W^{(2)}, h_1(W^{(1)}, h_0(W^{(0)}, x_i^{(0)}))), y_i). \quad (2.13)$$

If we were to perform backpropagation based on a single point (which implies our mini batch is of size 1 (if this is not the case then the final loss is the average over all losses for the points in the batch)), then we update each of the parameters via the chain rule, starting from the parameters in the end of the network and making our way back to the parameters in the beginning of the network. By way of illustration, if we want to find the partial derivative of the loss function L_i with respect to the parameter $W^{(2)}$, then using the chain rule we would get

$$\frac{\partial L_i}{\partial W^{(2)}} = \frac{\partial L_i}{\partial x_i^{(3)}} \frac{\partial h_2}{\partial W^{(2)}}. \quad (2.14)$$

See Fig. 2.2 for another example of computing the partial derivatives using forward and backward passes performed on scalar functions (*i.e.*, which produce scalar values as opposed to vectorized values).

2.2 Convolutional Neural Networks

We will now discuss the different components of Convolutional Neural Networks (CNNs) which will help to understand how these networks achieve state-of-the-art results in image classification tasks.

2.2.1 Convolutions

Convolution is a linear operation performed between two functions, either continuous or discrete, though we will only touch upon the discrete case which is relevant to 2D image grids. The two functions on which the convolution is performed are referred to as a signal (*e.g.*, an image) I and a kernel (AKA filter) K and is noted as $I * K$. More formally, in the 1D case we have:

$$I[x] * K[x] = \sum_{k=-\infty}^{\infty} I[k] \times K[x-k]. \quad (2.15)$$

Since images are stored in 2D (and 3D) pixel grids, we use 2D convolution, defined as:

$$I[x,y] * K[x,y] = \sum_{k_1=-\infty}^{\infty} \sum_{k_2=-\infty}^{\infty} I[k_1][k_2] \times K[x-k_1][y-k_2]. \quad (2.16)$$

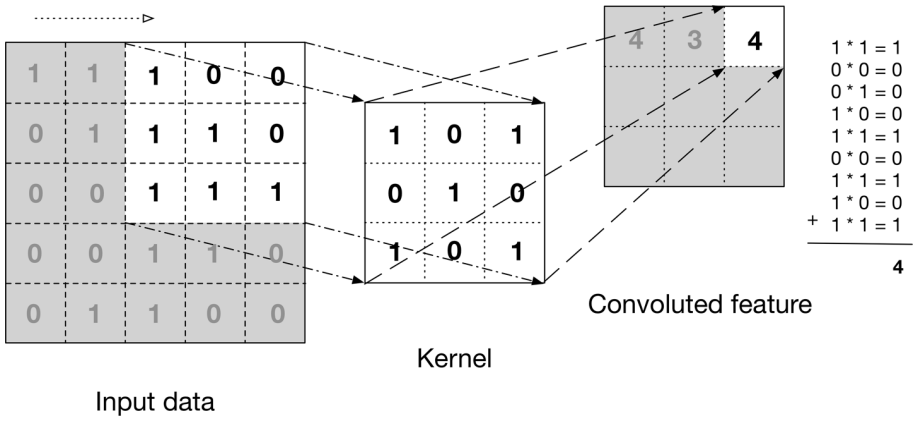


Figure 2.3: The convolution operation when performed on an image I and kernel K. Image credit: www.davidsbatista.net/

In the 2D convolution operation the kernel K is slided over different locations in I starting at the top left location, moving left to right and going downwards. In each of these locations, K is placed directly over a group of pixels in I which have the same size and shape of K (implying that K must lie completely inside the borders of I. Further below we discuss certain implications of this regarding padding and downsizing). For each of these locations an element-wise multiplication operation is performed between K and the corresponding pixels in the chosen location in I. The output of the convolution at that location is the summation over the values returned by the previous multiplication operation. See Fig. 2.3 for a good visualization of the 2D convolution operation. (Note that in the above equations there is also a flipping operation performed on the kernel, prior to the kernel image multiplication. Namely, first the kernel is flipped in the horizontal and vertical directions and only then is the above multiplication performed (*e.g.* for a given location x in the 1D case , $K[1]$ is multiplied by $I[x-1]$ and $K[-1]$ is multiplied by $I[x+1]$). That being said, since in the case of deep learning the kernel parameters are learned during the training phase, it can be assumed that the learned kernels have already been flipped).

Convolutions are used in practice in order to locate certain features in the images, such as edges, and that is how they help CNNs obtain state-of-the-art classification results.

2.2.2 Convolutional Neural Networks

A disadvantage of a completely FC network (*i.e.*, a network which is constructed only of FC layers) is the space complexity required in order to store all the parameters which define the different neurons, and the time complexity required to learn those parameters. This is due to the fact that in order to produce a single neuron at a certain layer there is a need to learn a corresponding weight vector the size of the previous layer ($+1$ for the bias parameter). Therefore, if, for example, there are n neurons in layer i and m neurons in layer $j = i + 1$, then in order to calculate the value of neurons in layer j we need to learn a weight matrix the size of $m \times (n + 1)$ (where we add the bias parameter b as another column in the weight matrix w and add a homogeneous coordinate to x so that the dimensions would fit). Since we are dealing with images, the input layer alone would naturally have thousands or even millions of entries and thus the first weight matrix would be huge. This is without taking into consideration the fact that we may have many hidden layers, each containing many neurons for which we would need to learn parameters.

We should also note that images are usually stored in a $H \times W \times C$ pixel grids, where H stands for height, W stands for width, and C stands for channel (*e.g.*, a RGB image would have three channels). Having this in mind, another disadvantage with the FC networks is that they do not exploit all of the information that the structure of the image holds. There is a strong correlation between neighboring pixels and between pixels at the same pixel location in the different channels. For example, if we have a pixel location not on the borders at location $[x, y, z]$ in the RGB image grid, we would expect the pixel values at the immediate neighboring locations (*i.e.*: $[x-1, y, z], [x+1, y, z], [x, y-1, z], [x, y+1, z], [x-1, y-1, z], [x-1, y+1, z], [x+1, y+1, z], [x+1, y-1, z], [x, y, z-1], [x, y, z+1]$) to be strongly correlated with the pixel values at $[x, y, z]$. This correlation decreases exponentially as the distance grows. The linear operation we previously used to create the neuron in the FC layer does not exploit these important relations between pixel locations.

A CNN is an ANN which specifically tackles the FC problems mentioned above, and it does so via convolutional layers. These layers perform the convolution operation on the previous layer, using kernels, *i.e.*, a group of small learnable weight matrices which are shared during the creation of many of the neurons in the following layer. In addition to this weight sharing feature which substantially decreases the number of parameters needed to be learned, each neuron is created via a weighted sum of pixel values at neighboring

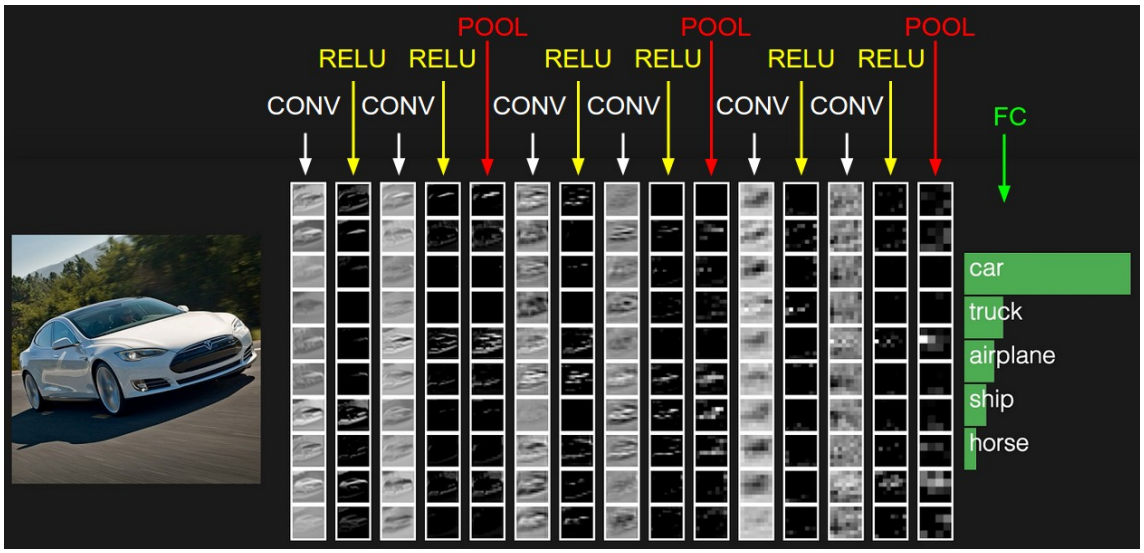


Figure 2.4: A classic CNN architecture. The different feature maps generated after each conv layer are also visible. Image Credit: cs231n.github.io

locations, thus exploiting the important surrounding information.

The general layout of the CNN architecture which performs multi-class classification is as follows: Input image $\rightarrow$ $\{$(Convolution + Relu)$\times t_1 \rightarrow$ Pooling layer$\}$ $\times$ $t_2 \rightarrow$ (Flatten + FC layer) $\rightarrow$ Softmax (output layer) . Were t_1 and t_2 are integers greater than zero describing the amount of occurrences of sub modules of the CNN. Relu is the non-linear activation function we saw earlier. The pooling layer (AKA down-sampling layer) is a way to decrease the size of the input, and is usually performed by either max-pooling, or by taking a bigger stride size in the convolution operations. The intuition behind the need of dawn-sampling operations is that (as we will describe in the next section) the conv layer helps us find features which best represent the image we are classifying (see next section below). Therefore, if it is known that a certain feature is present in a certain location in the image, then the resulting classification decision should not be affected if that feature would move a pixel or two in either direction. Thus downsampling can allow us to discard the unimportant information, save time spent on duplicate calculations, and focus solely on those features that are important (and as noted in section 2.5.1 this gives the network a certain degree of invariance to translations).

In the context of deep learning there are a few interesting points regarding convolution that we need to discuss:

- **Zero padding**

Assume we have an image I of shape $n \times n$ and we perform on it a convolution with a kernel K of shape 3×3. A question that comes up is what do we do with the pixels in the borders of I? If we were only to perform the convolution for the indexes in I on which we could place the center of K so that all other indexes in K are multiplied by existing indexes in I, then we would not perform convolutions for the values in the borders of I. This means the matrix we end up with is of size $(n-2) \times (n-2)$. If on the other hand we want our output matrix to be the same size as the input matrix, then we would need to add some "padding" around the matrix I so that we could perform convolutions for the original borders of I. For the case described above we would add a single row of zeros on top of I, a single row of zeros on the bottom of I, a single column of zeros on right of I and a single column of zeros on left of I.

- **Stride size**

  In the convolutions we talked about previously, once we performed the convolution at location $I[x, y]$ with the kernel K, we then would move one location to right to $I[x+1, y]$ (assuming x is not on the right border) and perform the convolution there. This way we would iterate over every row in I performing convolutions on each entry in I, moving from left to right. This is referred to as stride size $= 1$. Another option which is often used in CNNs is to use a stride size different than 1, such as 2. The intuition here is similar to that described when we discussed the pooling layer, namely once a feature is located, the exact location does not matter. For the lower levels of the network where the input is relatively large and where a small pixel movement will not affect the overall feature detection, we could save a lot of the expensive computation time by down-scaling the problem.

- **Window size and 1×1 convolutions**

  CNNs typically use kernels for which the height and the width have the same size. The choice of the exact kernel size is up to the network's architect, where common sizes in use are 3×3, 5×5, 7×7 or 9×9. Though using kernels with size 1×1 might seem to be an odd choice, it turns out that these convolutions are quite useful in practice. As previously noted, pooling layers and different stride sizes are useful techniques in order to reduce the training time. There is another issue which arises

with regarding to CNN training, which is that the deeper we go in the network, the more kernels we generally find, and so amount of the feature maps produced by those kernels increases with the depth of the network. Since we convolve each kernel with every entry in every feature map, this is quite expensive computationally. Another issue regarding the feature maps depth is that in modern networks (such as Google's Inception or in DenseNet which we will later touch upon) there is a feature map concatenation stage, meaning that in a certain stage of the training we append the newly learned feature maps with the older ones, and theoretically this could cause the feature maps and the learning time to explode as the network goes deeper. In order to reduce the size of the feature maps depth, 1×1 convolution is used. Previously we discussed how to perform convolution given a 1D or a 2D signal. When the feature maps we perform convolution on are in 3D (such as RGB images), we iterate over the 2D feature maps constructing the depth, and convolve each of the 2D feature maps separately. Then we create a single 2D feature map by summing the corresponding (i, j) entries at each feature map created during the convolution. Having this in mind, 1×1 convolutions (with padding$= 0$) can reduce depth of the feature maps to a single 2D feature map without changing the width or the height. This has the effect of summarizing the different information in the different feature maps and maintaining the important information.

2.2.3 Feature Maps and Feature Extraction

The idea in deep learning in general, and specifically when using CNNs, is that we would like to learn good feature extractors (*i.e.*, kernels) that help us find the different features which describe the image in the best way. We can then take the features that were generated and try to classify the original image based on them. Each convolution operation applied on a kernel and on a layer of neurons (which are basically a group of learned features), creates a new set of features which we call "feature maps".

Receptive Field. If convolution is performed directly on the input images, then each neuron in the output of the first conv layer is generated by a weighted sum between a pixel at some location in the original image and the pixel's surrounding neighbors. Even if these generated neurons in the first layer are only generated by a very small portion of the original image, in the second conv layer each neuron would be generated by a group

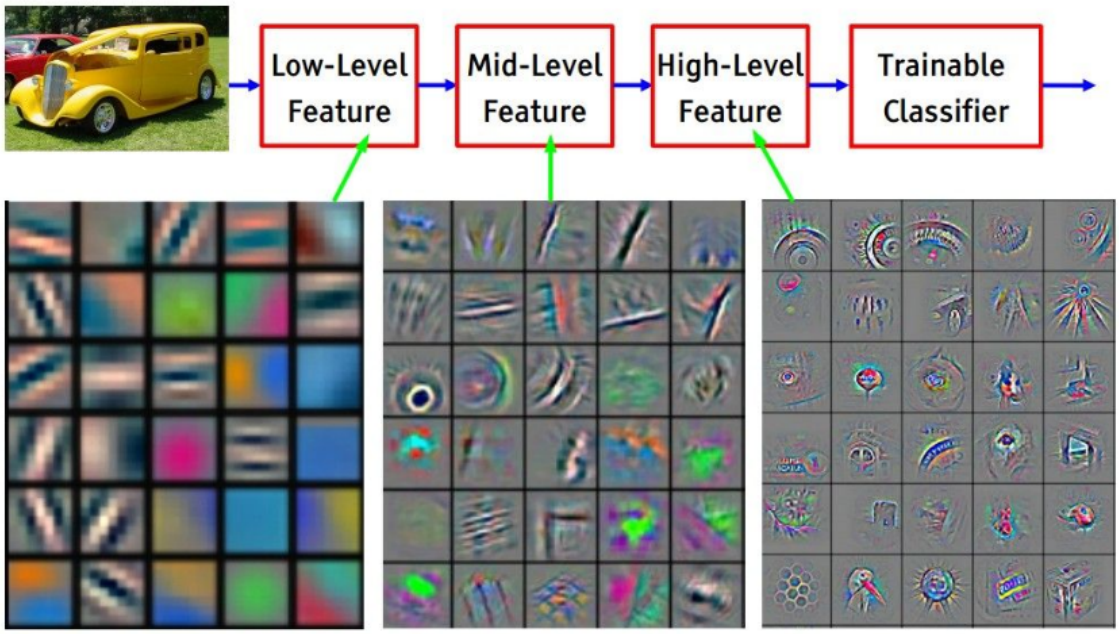


Figure 2.5: Features generated in different levels of the CNN represent various levels of abstraction. We see that deeper we go in the network, the kernels generated in those layers can extract more abstract features. Credit: https://arxiv.org/abs/1311.2901

of neurons in the first layer, and thus they are generated by all the pixels which generated those neurons in the first layer. As we move further into the network, each neuron is affected by a larger and larger portion of the image. The amount of pixels in the input image each neuron is affected by is referred to as the neuron's "Receptive Field".

The term Receptive Field is borrowed from biological vision and refers to a characteristic of neurons in the brain's visual cortex. Similar to its use in deep learning, the original term refers to a specific region in the visual space which causes a stimulation of neurons which produce a visual response in the cortex. There are different types of neurons in the visual cortex. There are simple neurons which get triggered by visual activity in very small regions of the visual space. If many of these simple neurons get triggered by a visual stimulus, they can send a message to a neuron which gets triggered only after a group the simple neurons fire. This more complex neuron will have a receptive field composed of the receptive fields of the simpler neurons, and the activity which will cause it to fire would now be more complex. If a group of these more complex neurons are all connected to another neuron to whom they all send a message, then this neuron has an even bigger receptive field and would get triggered by a more complex visual stimuli. By way of illustration, suppose the first simple neurons fire in the presence of a dot in a very

small location in the visual space. If brought together, these small separate dots can form a line or an edge, and if all these neurons are all connected to another neuron, this neuron might react to only a certain type of edge in a certain location. Next, several of such edge neurons may come together to create a more complex shape such as an eye in a certain location, and if these neurons which find shapes such as eyes come together with neurons which find a nose, a mouth and an ear in specific locations, then the next level neuron has enough information to possibly fire in the presence of a face.

Interestingly, it turns out that a very similar scenario occurs during the CNN learning process. When visualizing the different feature maps generated in the different conv layers in the CNN, it was shown by Zeiler *et al.* [55] that during the learning process the different conv layers learn different levels of feature representation. They show that the first conv layer's kernels are responsible for finding simple features, such as different types of edges or colors, and the deeper the conv layer is in the network, the more abstract the features that it will be able to locate are (see Fig. 2.5).

2.3 Very Deep Networks

2.3.1 Introduction to Very Deep Nets

Based on what we have previously seen (section 2.2 above), the consensus in the field of deep learning is that the more conv layers a network has, the more abstract features the network can obtain of the input images, which would have the potential to help extract important features for the classification task at hand. We also noted the advantages of using conv layers as opposed to FC layers with regard to the number of parameters, which would have a direct impact on the computation time and on the space complexity required. Due to the success of CNNs in many tasks and specifically in classification tasks, researches have tried over the last few years to add more and more conv layers to their networks in order to achieve state-of-the-art classification results. Presented below is a short overview of CNN's achievements over the last few years in classification tasks, with specific reference to those network's architectures and length (with regarding conv layers).

The recent upsurge of deep networks began in the 2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC). ILSVRC is a computer vision competition which was held between the years 2010-2017. The competition was constructed of several different

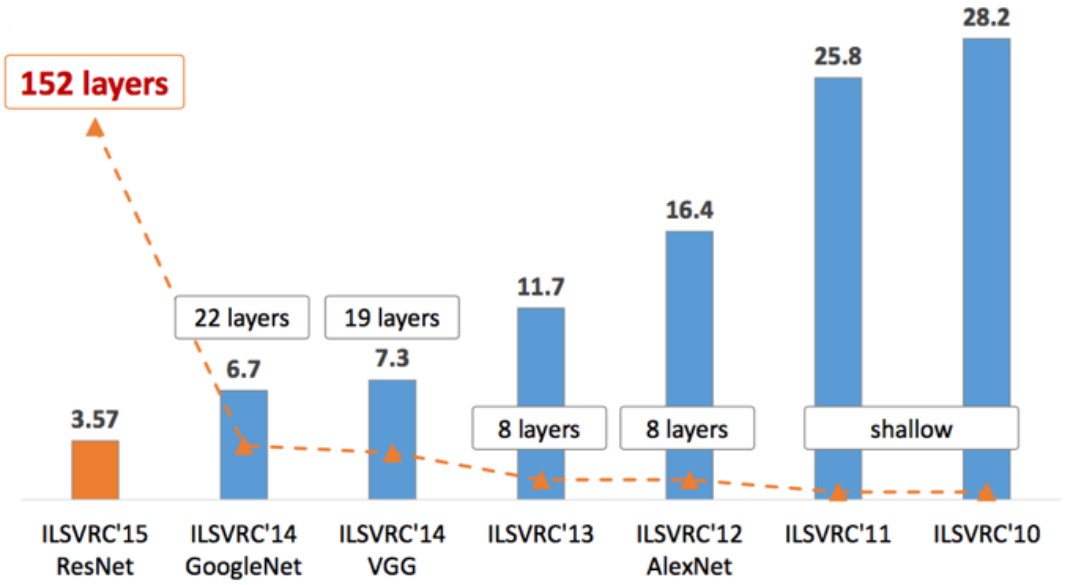


Figure 2.6: CNN depths in the ILSVRC competition over the years. Image Credit: kaiminghe.com

challenges on the ImageNet dataset, which is composed of more than 14 million annotated images belonging to 20,000 different categories. In the 2012 competition, AlexNet [23] surpassed all the other methods in a classification task (with an error rate of 15.3%, second place error rate was 26.2%). AlexNet is a CNN which consists of 5 conv layers followed by another 3 FC layers. Since adding layers allows learning more features at various abstraction levels [55], many works focused on getting deeper networks. At the 2014 ILSVRC competition, the top two ranked networks, the VGG[46] and the GoogleNet [48] networks, were the deepest networks of their time, constructed of 19 and 22 conv layers respectively. Using the number of conv layers in the VGG network as a lower bound, we define *very deep networks* as networks with at least 19 conv layers.

2.3.2 ResNet

In the previous chapter (section 2.1.1 above) we examined popular activation functions. We related to the sigmoid function and mentioned that it is not frequently used these days as an activation function. The reason for not using the sigmoid function as an activation function is linked to the specific derivative of the sigmoid function, which imposes a problem during backpropagation. As already discussed with respect to backpropagation (section 2.1.4 above), backpropagation is a mechanism of updating the network's parameters by using the

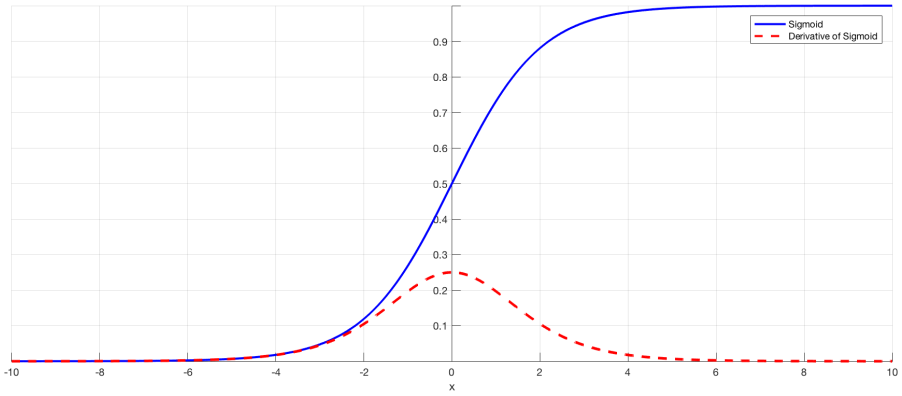


Figure 2.7: Sigmoid and its derivative. We can see that for most values of x, the derivative is close to zero. Image Credit: isaacchanghau.github.io

chain-rule. Starting from the output of the final layer and going backwards, we compute the partial derivative of each parameter in the network by calculating the derivatives along the way and using the chain-rule. Therefore if we were to use sigmoid as our activation function then basically for every neuron we will have to compute the derivative of sigmoid, and multiply it by the incoming gradient. The derivative of the sigmoid function is:

$$\sigma'(x) = \sigma(x) \times (1 - \sigma(x)). \tag{2.17}$$

This derivation reaches a maximum of 0.25 when $x = 0$ and quickly decreases as we move away from 0 on either direction (see Fig. 2.7). What this means is that for most values of x we will get a derivative which is very small, and since during backpropagation we multiply many of such derivatives with each other, we will end up with extremely small numbers exponentially fast. By the time we would get back to the parameters needed to be updated, the gradient which we would get would be basically zero and thus the parameter updates would be non-existent. This prevents the network from learning the optimal parameters during backpropagation. The problem in which the gradient flowing backwards becomes zero is referred to as the 'vanishing gradient problem'. For many years the vanishing gradient problem has prevented networks from going deeper. This is because that even if we choose our activation functions wisely (*e.g.*, using the relu function), since we have many layers and during backpropagation the different derivatives get multiplied with each other, then even if a few neurons produce a relatively small gradient, this gradient will very quickly go to zero.

In order to handle the vanishing gradient problem and allow networks to go deeper,

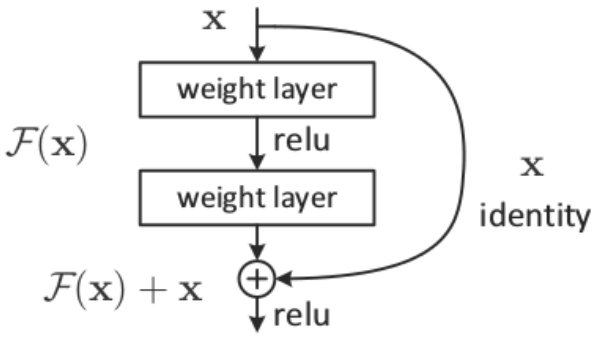


Figure 2.8: Resnet's skip connection. Image Credit: [13]

He *et al*. [13] introduced the Residual Network (ResNet) and produced a framework that enables creating networks with tens and hundreds of layers. They managed to avoid the vanishing gradient problem by using skip connections, *i.e*., feature summation of the form:

$$x_l = H(x_{l-1}) + x_{l-1} \tag{2.18}$$

(see Fig. 2.8). What this means is that during backpropation, the incoming gradient entering a neuron can "skip" the need of going through another multiplication of derivatives, and instead can go straight back to input x_{l-1} (which created the following neuron $H(x_{l-1}) + x_{l-1}$ during the forward pass). Once all the different paths of backpropagating derivatives flow back to the input x_{l-1}, the value of the input's derivative is the sum of all the incoming derivatives, and so even if one of them has a value close to zero, it would get canceled out by the bigger value and thus prevent the gradient from vanishing.

2.3.3 DenseNet

Several works that built on ResNet [18, 50, 53] improved on the original network's performance and achieved superior classification results. A recent variation of ResNet, DenseNet [17], approached the vanishing gradient problem via feature map concatenation of the form

$$x_l = H([x_0, x_1, ..., x_{l-1}]), \tag{2.19}$$

thereby performing feature reuse. This way the creation of a new feature map in the network literally takes into account every single feature map created up to that point. By doing so, state-of-the-art results on multiple datasets were achieved. To prevent the amount of features needed to be stored from exploding and the convolution time that these feature

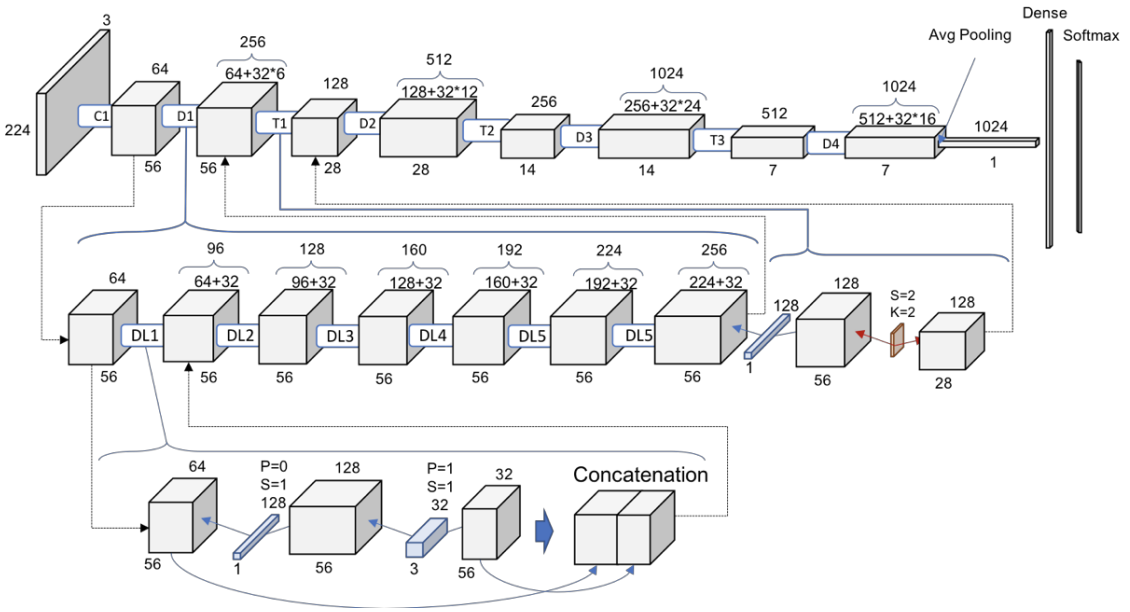


Figure 2.9: A visualization of the DenseNet architecture. Image Credit: https://towardsdatascience.com/@pabloruizruiz

maps would require, DenseNet reduces the amount of feature maps by adding bottleneck layers which use 1×1 convolutions to reduce the depth of the input before performing 3×3 convolutions on the input. DenseNet is composed of DenseBlocks, where each block contains consecutive conv layers and the feature maps dimensions remain constant between the layers (see Fig. 2.9). After each conv layer, K new feature maps are concatenated, where K is referred to as the network's growth factor. In order to handle the down sampling between the blocks, transition layers are placed between them.

2.4 Image Warping

The following is a discussion of how image transformations are performed in practice. This is needed in order to understand STNs and to later understand our proposed alignment method.

Before diving into spatial transformations, there is a need to understand how images are represented. An image I, which is stored in a pixel grid of size $W \times H \times C$ (where W stands for width, H stands for height and C stands for channel), can be represented as a discrete function which takes as input w, h, c values where

$$w \in \{-\frac{(W-1)}{2}, -\frac{(W-1)}{2} + 1, ..., 0, ..., \frac{(W-1)}{2} - 1, \frac{(W-1)}{2}\}, \tag{2.20}$$

$$h \in \{-\frac{(H-1)}{2}, -\frac{(H-1)}{2}+1, ..., 0, ..., \frac{(H-1)}{2}-1, \frac{(H-1)}{2}\}, \tag{2.21}$$

$$c \in \{0, 1, 2, ..., C-1\}, \tag{2.22}$$

and returns a brightness intensity value (*i.e.*, pixel value) located at $I[w, h, c]$. It is noteworthy that in this setting the origin of the each 2D pixel grid in each channel of I is placed in the middle of the height and width, and that indexes in the vertical line increase as we go downwards. Another common setting is to find the origin of each 2D grid at the top left corner. Here we will work only with the first setting.

2.4.1 Spatial Transformations

Suppose we have an image I_s (the source image) to which we want to perform a certain transformation in order to reach a different image I_t (the target image) which is generated after performing a transformation on I_s. This is also referred to as image warping. In 'spatial transformations', the value at each pixel location $I_s[w, h, c]$ is mapped to a possibly different location, forming I_t (which is a different image unless the transformation we apply is the identity transformation, in which case $I_t = I_s$) while each pixel location at I_t corresponds to at least one other pixel location at I_s.

There are many different possible spatial transformations which one can use in order to perform image warping (e.g., affine, homographies, thin-plate-splines, diffeomorphism etc.). Here we will focus mainly on affine transformations as these are the transformations that we used during our alignment experiments.

2.4.2 Affine Transformations

An affine transformation is a transformation based on the affine function:

$$f(x) = Ax + b. \tag{2.23}$$

Note that this function is not linear (with respect to x) since the origin is also shifted when $b \neq 0$. All linear functions of the form Ax are also affine functions for which b is set to 0. Let us look at one of the 2D grids composing one of the channel grids of some image I. We need to map every 2D point to a new location. For a certain point $p := [\begin{smallmatrix} x \\ y \end{smallmatrix}]$ in this 2D grid, there are a few different transformations which compose affine transformations

and which can be applied to this point. Since we are transforming a 2D point, the output of the transformation should also be a 2D point. Let us next consider a transformation matrix $M \in \mathbb{R}^{2x2}$. In order to apply a transformation on point p, we would perform matrix multiplication of the form Mp. This of course is a linear transformation. For example, if $M = \left[\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right]$ (*i.e.*, the identity matrix), then $Mp = p$, and this transformation matrix will keep every point in its original location. In order to transform the entire image I using M, we would take a vectorized representation of the image grid:

$$\left[\begin{bmatrix} -\frac{(W-1)}{2} \\ -\frac{(H-1)}{2} \end{bmatrix}, \begin{bmatrix} -\frac{(W-1)}{2}+1 \\ -\frac{(H-1)}{2} \end{bmatrix}, ..., \begin{bmatrix} \frac{(W-1)}{2} \\ \frac{(H-1)}{2} \end{bmatrix}\right], \tag{2.24}$$

and multiply it from the left by M, thus generating a new location for each of the pixel values in I. An examination of what is done when the new location of a pixel value falls in a non-integer point appears further below.

There are different transformations which together compose affine transformations.

- Scaling. There are two cases to consider in regard to scaling:

  1. For values $a, d \geq 0$, if we set $M = \left[\begin{smallmatrix} a & 0 \\ 0 & d \end{smallmatrix}\right]$ then $M = \left[\begin{smallmatrix} a & 0 \\ 0 & d \end{smallmatrix}\right] \times \left[\begin{smallmatrix} x \\ y \end{smallmatrix}\right] = \left[\begin{smallmatrix} ax \\ dy \end{smallmatrix}\right]$, this transformation scales the input image. If a or $d < 1$, then the transformation will shrink the the image in the width or height of the image respectively. If a or $d > 1$ then the transformation will enlarge the the image in the width or height of the image respectively.

  2. For values $a, d < 0$, we would get the mirror image of transformation obtained by $-a, -d$ (which is the positive case previously discussed).

- Rotation. In order to rotate the image around the origin we use the following transformation matrix: $M = \left[\begin{smallmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{smallmatrix}\right]$.

- Shear. In this mapping, the x coordinate is increased/decreased by a value proportionate to the y coordinate and the y coordinate is increased/decreased by a value proportionate to the x coordinate. The matrix for this mapping is defined $M = \left[\begin{smallmatrix} 1 & b \\ c & 1 \end{smallmatrix}\right]$ where $b, c \in \mathbb{R}$. When applying this matrix on a certain point, the result is: $M \times \left[\begin{smallmatrix} x \\ y \end{smallmatrix}\right] = \left[\begin{smallmatrix} x+by \\ y+cx \end{smallmatrix}\right]$.

Given the above transformations, let SC denote the scaling matrix, R denote the rotation matrix, and SH denote the shear matrix. Suppose that these transformations are performed

on the image in a certain order, such as transforming each index p by performing the following transformations $SC \times R \times SH \times p$. Since all of the above mappings are merely matrix vector multiplications of the form $M \times p$ and since matrix multiplication is associative, we can first compose all of these transformation matrices together to create a single transformation matrix T, and then we can simply compute $T \times p$ (see Fig. 2.10) . The foregoing enables us to summarize the above linear transformations in a single matrix $T \in \mathbb{R}^{2x2}$ of the form $\left[\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right]$. Note that since matrix multiplication is not commutative there is a need to keep the same order of the performed transformations.

- Translation. Affine transformations have the nonlinearity property due to the translation transformation which is defined as $\left[\begin{smallmatrix} x \\ y \end{smallmatrix}\right] + \left[\begin{smallmatrix} e \\ f \end{smallmatrix}\right]$ where $e, g \in \mathbb{R}$. In order to perform translations via matrix multiplication we define the transformation matrix $M = \left[\begin{smallmatrix} 1 & 0 & e \\ 0 & 1 & f \end{smallmatrix}\right]$ and add a homogeneous coordinate to p, creating $p' := \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$. When applying the transformation we get: $M \times p' = \left[\begin{smallmatrix} x \\ y \end{smallmatrix}\right] + \left[\begin{smallmatrix} e \\ f \end{smallmatrix}\right]$. In order to allow matrix concatenation of many different affine transformations as before, we append a new unit row at the bottom of M, creating the combined matrix $\begin{bmatrix} a & b & e \\ c & d & f \\ 0 & 0 & 1 \end{bmatrix}$. In order to generate the final point $p \in \mathbb{R}^2$ we simply remove the final homogeneous coordinate.

To summarize, the family of affine transformations can be defined by six parameters a, b, c, d, e and f. Different tranasformations would require different number of parameters (*e.g.*, homography transformations are defined by eight parameters). Our work relates only to affine transformations.

2.4.3 Backward Warping

After choosing a certain transformation family defined by a group of parameters θ (in our case the transformation family is the affine transformation defined by 6 parameters as we have previously seen), it is possible to perform image warping. Image warping is simply the process of transforming each pixel location in the source image to a possibly new location.

There are two possible methods which can be used in order to perform the warping. One is 'forward warping', and another is 'backward warping'. Forward warping is very intuitive. By way of illustration, take the source image I_s, and apply the transformation on every pixel location in the image grid in order to generate the target image I_t. Though this

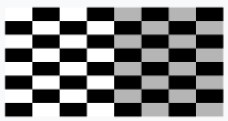
Transformation name	Affine matrix	Example
Identity (transform to original image)	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	
Reflection	$\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	
Scale	$\begin{bmatrix} c_x = 2 & 0 & 0 \\ 0 & c_y = 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	
Rotate	$\begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$	where $\theta = \frac{\pi}{6} = 30°$
Shear	$\begin{bmatrix} 1 & c_x = 0.5 & 0 \\ c_y = 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	

Figure 2.10: The different linear transformations which construct affine transformations and which can be summerized in a 2×2 transformation matrix via four parameters. If we would like to add the translation transformations we would need to add another two parameters forming a 2×3 (or 3×3) transformation matrix seen above (the translation transformation is omitted from this figure). Notice how all affine transformations preserve parallel lines. Image Credit: Wikipedia.org

approach is very straight forward, it can easily output an undesired result. As mentioned above, in image warping each pixel location $[w, h]$ in I_s is mapped to a (possibly) different location. In the forward mapping procedure, two different pixel locations can fall into the same location in I_t. This implies that there exists some entry in I_t to which no pixel location from I_s is mapped. In this case, the locations in I_t which are not in the range

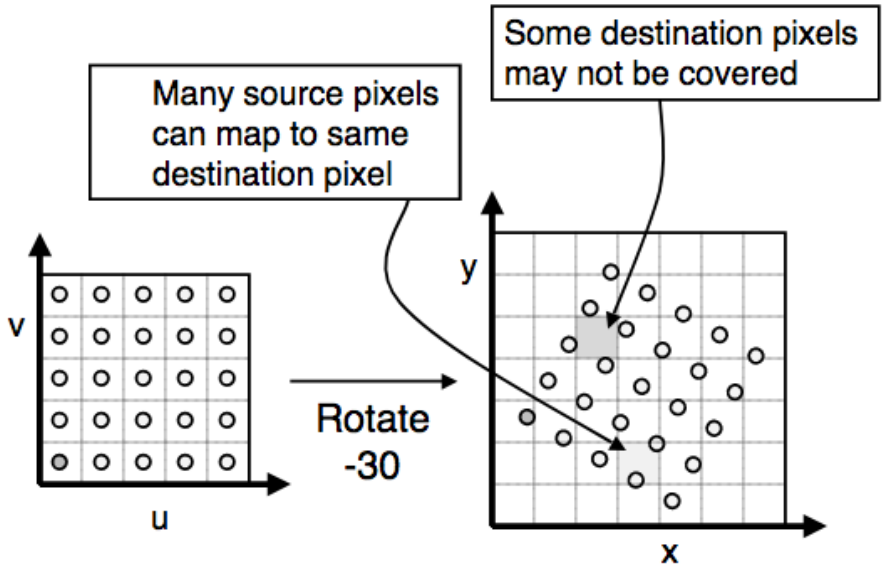


Figure 2.11: Forward warping can form holes in the middle of the transformed image. This issue is solved by performing backward warping. Image Credit: /www.cs.princeton.edu/courses/archive/fall00/cs426

of the transformation's mapping would have an undefined pixel value, which is usually filled in with black pixels. This scenario would leave the resulting image with undefined patches of black holes in different locations in the middle of the I_t, which is undesirable as it is preferable for the pixel values to be continues (*e.g.*, we wouldn't like there to be an unrelated black hole in the middle of our grandmother's sweater).

The second approach to image warping is called backward warping. This technique is a bit less intuitive but overcomes the previous issue. In backward warping, we view the parameters of the transformation learned as belonging to the inverse transformations applied on I_t in order to go back to the original image I_s. Therefore, instead of applying the learned transformation to each index in I_s, the transformation is applied to each pixel location in the output grid, which will take the pixel location from $I_t[w', h']$ to a (possibly) different location which might fall within I_s's image grid. Assuming the chosen transformation family is continuous (which affine transformation are), it is no longer possible that in the resulting I_t there will be a black hole in the middle of the image. Since the transformation is continuous, a scenario of 3 neighboring pixel locations a, b, c in I_t, where a and b are both mapped to locations inside the pixel grid of I_s but c (which is located between a and b in I_t) is mapped to a location outside of I_s (a scenario resulting in a black hole between a and b), is not possible in such case of continuous transformations.

What may occur even in backward warping, is that different borders of the resulting I_t will be black. An example where this may occur is if in order to get from I_s to I_t we move every pixel value in I_s one pixel location to the left (*i.e.* we apply a small translation transformation). In this case, the transformation which is learned translates each pixel location in I_t one pixel to the right (as it is the inverse transformation), and thus every index in the right most column of I_t will be mapped to points outside of I_s's image grid which is undefined, resulting in that column being filled in with black pixels. This behavior is expected in image warping (since grandma's sweater now remains intact).

2.4.4 Bilinear Interpolation

There is a crucial issue with image warping (both forward and backward) which requires further discussion. Recall that images are discrete functions which return a pixel value only for predefined discrete locations in the image grid. Suppose now that our learned transformation was applied on a certain location $[w', t']$ in I_t (in the case of backward warping) which is then mapped to a different location $[w, t]$ inside I_s, being a location that is not discrete. In this case $I_s[w, t]$ is not defined, and therefore the question of which pixel value should be stored in $I_t[w', h']$ needs to be addressed. For this reason 'image interpolation' is performed. Interpolation is a general term referring to the process of guessing which pixel value should be used in a certain location. There are a few common methods for performing such interpolation. Here we will focus mainly on a method called 'bilinear interpolation' which is the most common technique, but other methods may be used as well (*e.g.*, nearest neighbor).

Before exploring bilinear interpolation, linear interpolation will be briefly reviewed. Given two discrete neighboring locations x_1, x_2 (so that $x_2 - x_1 = 1$), which are located on the real line (thus the 'linear' part of the name) and which hold pixel values v_1, v_2 respectively, we can interpolate the value of another point p where $x_1 \leq p \leq x_2$. This value is interpolated via its immediate neighboring values v_1, v_2. The nearest neighbor approach is choosing the value for p (which we will note as val_p) to be the value closest to p's location, *i.e.*, either v_1 or v_2. In the linear interpolation approach, we will choose val_p to be the value of the weighted sum of the p's immediate neighbors x_1, x_2, in the following

way:

$$\text{val}_p = \frac{|x_2 - p|}{x_2 - x_1} \times v_1 + \frac{|x_1 - p|}{x_2 - x_1} \times v_2. \tag{2.25}$$

Notice that if the distance between p to x_1 is d (and so the distance between p to x_2 is $(1-d)$) then the weight which we will give x_1 in the linear interpolation approach is $(1-d)$. What this means is that if p is closer to x_1 than to x_2 (*i.e.*, $d < (1-d)$) then the weight of x_1 is greater than the weight of x_2. The extreme case is when $p = x_1$ (or $p = x_2$) in which case val_p is v_1 (or v_2) as expected.

In bilinear interpolation, the task is to find the value of a 2D point $p = [p_1, p_2]$ given its four immediate discrete neighboring locations in a 2D grid are $x_{tf} = [x_1, y_1], x_{tr} = [x_2, y_1], x_{bf} = [x_1, y_2], x_{br} = [x_2, y_2]$, for which $x_1 < x_2$ and $y_1 < y_2$ (*i.e.*, x_{tf} is the top left neighbor of p, x_{tr} is the top right neighbor of p, x_{bf} is the bottom left neighbor of p and x_{br} is the bottom right neighbor of p), and given the corresponding pixel values in those locations are v_1, v_2, v_3, v_4. As prior discussed, had we applied the nearest neighbor approach, we would simply have returned the value of the location which is closest p, and as before, in bilinear interpolation (which is based on linear interpolation) we will return the weighted sum of the values in the four immediate neighboring locations. Only now, in order to calculate the weighted sum, there is a need to first perform twice a linear interpolation in either one of the axes (either the horizontal or the vertical axes) and then perform once more a linear interpolation in the remaining axis (vertical or horizontal correspondingly) as follows:

$$H_1 = \frac{|y_2 - p_2|}{y_2 - y_1} \times v_1 + \frac{|y_1 - p_2|}{y_2 - y_1} \times v_3, \tag{2.26}$$

$$H_2 = \frac{|y_2 - p_2|}{y_2 - y_1} \times v_2 + \frac{|y_1 - p_2|}{y_2 - y_1} \times v_4, \tag{2.27}$$

$$\text{val}_p = \frac{|x_2 - p_1|}{x_2 - x_1} \times H_1 + \frac{|x_1 - p_1|}{x_2 - x_1} \times H_2. \tag{2.28}$$

Another way to view bilinear interpolation is that the weight that is given to a certain neighbor x_i ($i \in \{tf, rf, bf, br\}$) of p, is the area of the rectangle formed between p and the neighbor in the opposite location of x_i. (see Fig. 2.12). This way the closest neighbor gets the largest weight in the bilinear interpolation. Continuing the example from above, suppose the distance between p_1 to x_1 is d_1 and the distance between p_2 to y_1 is d_2. In this interpretation the value interpolated for p is: $\text{val}_p = v_1(1-d_1)(1-d_2) + v_2(d_1)(1-d_2) + v_3(1-d_1)(d_2) + v_4(d_1)(d_2)$.

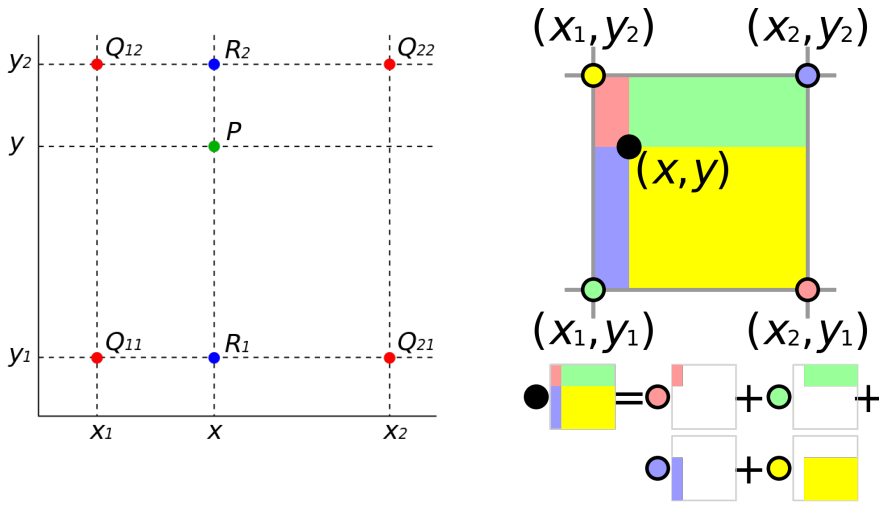


Figure 2.12: Two methods to perform bilinear interpolation. In the left image we perform two different linear interpolations in the horizontal axes and then another linear interpolation in the vertical axes. In the right image the weighted sum operation is easy to visualize, where each of the neighboring pixel values is multiplied by the weight of the opposite rectangle's area. Image Credit: Wikipedia.org

2.5 Spatial Transformer Networks

2.5.1 STN Introduction

Although CNNs have been proven to be a powerful classification tool, they are not invariant to different spatial transformations such as rotation, scaling, *etc*. To a certain extent, CNNs can cope with some simple spatial translations, but this is only due to the down sampling (*e.g.*, max-pooling) performed at different stages of the network. In addition these are merely a subset of all possible translations. To show this translation invariance, suppose we have a 2×2 pixel grid with pixel values located at entries $(0,0), (1,0), (0,1), (1,1)$ for which we will perform a 2×2 max-pooling stage. In addition, suppose all the transformations we permit are translations which move each pixel location one pixel in either direction (right, left, up, down) or one pixel diagonally in either direction (up-left, up-right, down-left, down-right) then there are eight possible such translations. Suppose without loss of generality that the maximum value is located at entry $(1,1)$. Due to max-pooling operation, there are exactly three out of the above eight translations which end up in (possibly) the same result after max-pooling. *i.e.*, left, up, up-left. Any other

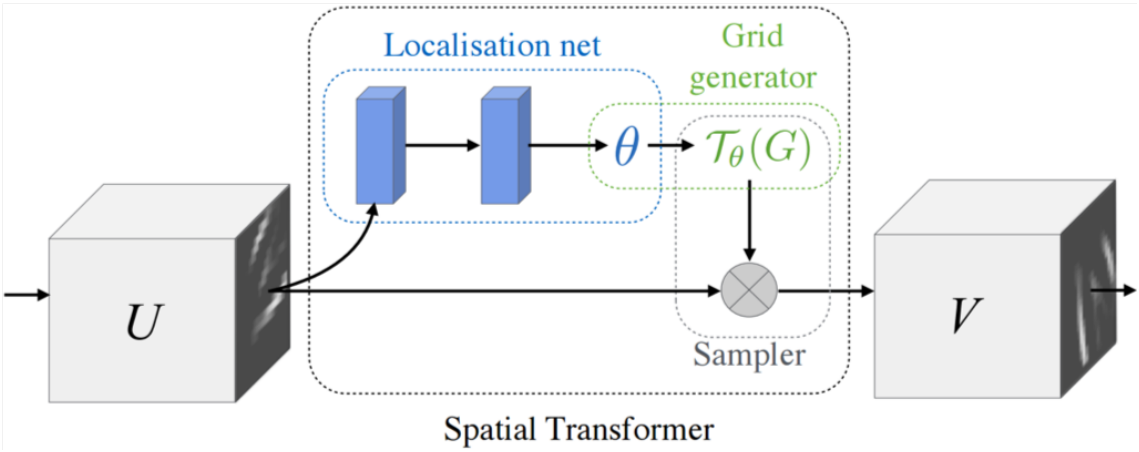


Figure 2.13: The three modules composing an ST layer. Image Credit: [21]

translation would remove the maximum point out of the 2×2 pixel grid. Even in this small subset of translations, it is possible that a new maximum will enter the grid from a different neighboring grid (as a result of the translations), and therefore the same value after max-pooling might not be produced. In light of the foregoing, we aim for CNNs to be invariant to different spatial transformations.

To address this problem, Jaderberg *et al*. created the Spatial Transformer Networks (STNs) [21]. An STN is a network which has a modular nature such that it can be incorporated in other networks, regardless of the task at hand. Therefore, we also refer to STNs as Spatial Transformer (ST) layers; namely, layers that learn and apply spatial transformations to their input such that they should help decrease the overall learning loss. It has been shown that applying an STN to a set of feature maps in any location in the network (namely the input images if the ST layer is placed directly after the input images; the ST layer can be placed in any location, such as after the feature maps generated by the third conv layer), can improve the network's classification performance [2, 30].

2.5.2 STN Architecture

This section explores the architecture of an STN and the different components which construct the ST layer.

Given a family of spatial warps (*e.g*., affine, homographies, thin-plate-splines, *etc*.), also known as transformations, parametrized by $\boldsymbol{\theta}$, the ST layer, proposed in [21], performs an input-dependent warp, $T^{\boldsymbol{\theta}}$, by regressing $\boldsymbol{\theta}$. The ST layer consists of 3 modules (see Fig. 2.13) :

1. Localization network. The first module of an ST layer is the localization network f_{loc}. This is a small network, usually with two or three hidden layers, which takes the input signal, $\boldsymbol{U}$, and learns the warp's parameters such that $f_{\text{loc}}(\boldsymbol{U}) = \boldsymbol{\theta}$. Any form of neural network architecture can be used for f_{loc} (such as conv layers), as long as the output layer is a Fully-Connected (FC) one with d neurons, where $d = \dim(\boldsymbol{\theta})$. By way of illustration, for affine warps in 2D, $d = 6$.

2. Parameterized grid generator. The parameterized grid generator creates a discrete regular 3D grid (*i.e.*, a lattice) of M points, (where M is typically the number of pixels in the input feature maps. Therefore, if the input features maps are the channels of the input image, then $M = H \times W \times C$): $G = (\boldsymbol{p}_m)_{m=1}^{M} \subset [-1, 1] \times [-1, 1] \times C$, points which are later transformed by the parametrized warp, $T^{\boldsymbol{\theta}}$. This grid will hold the transformed feature maps, after performing backward warping using the parameters learned by the localization network.

3. Differentiable image resampler. This module performs the backward warping using the output grid G and the parametrized warp $T^{\boldsymbol{\theta}}$ (this is done by a simple matrix multiplication), and it then performs interpolation generating the correct pixel value for each location in the output grid. The common interpolation method used is bilinear interpolation. This generates the output signal, $\boldsymbol{V}$, which is interpolated based on the warped points in $T^{\boldsymbol{\theta}}(G)$ while using as reference the input feature maps $\boldsymbol{U}$. Note that due to the need to resample the signal using interpolation schemes rather than just performing $\boldsymbol{V}_i = \boldsymbol{U}_i \circ T^{\boldsymbol{\theta}_i}$, it is necessary to also account for the resampling function during backpropagation.

2.5.3 STN and Backprop

As explained above, every operation performed during a deep learning forward pass must be differentiable, in order to later update the different parameters during the backpropagation stage, and an ST layer can be added at any stage in a different, global network. We introduced above three different components which construct this layer and examined different operations which are added to the general network during the learning stage. We now consider how these operations are indeed all differentiable.

The operation performed in the localization network is a simple regression with possibly some convolution operations in the middle. These are the bread and butter of any CNN and thus are clearly differentiable. This is true also with regarding to the backward warping

operation which is merely a matrix-matrix multiplication. The only operation remaining is that of the resampling function (*i.e*. the bilinear interpolation) which, as shown below, is also differentiable.

Let $\boldsymbol{U}$ be a certain input of feature maps and $\boldsymbol{U}^c$ be the 2D pixel grid of one of $\boldsymbol{U}$'s channels. Let (x_i^s, y_i^s) denote the location in $\boldsymbol{U}^c$ which a coordinate $i \in [1, ..., H \times W]$ from G^c is mapped to by $T^{\boldsymbol{\theta}}(G^c)$. (x_i^s, y_i^s) is then interpolated to produce a pixel value $\boldsymbol{V}_i^c$. Assuming we are using the bilinear interpolation method (as we did in our work), the kernel which computes $\boldsymbol{V}_i^c$ is:

$$\boldsymbol{V}_i^c = \sum_n^H \sum_m^W \boldsymbol{U}_{nm}^c \text{max}(0, 1 - |x_i^s - m|)\text{max}(0, 1 - |y_i^s - n|). \tag{2.29}$$

Now we define the partial derivative for the above kernel using sub-gradients (as the above maximum operation has a discontinuous point):

$$\frac{\partial \boldsymbol{V}_i^c}{\partial \boldsymbol{U}_{nm}^c} = \sum_n^H \sum_m^W \text{max}(0, 1 - |x_i^s - m|)\text{max}(0, 1 - |y_i^s - n|), \tag{2.30}$$

$$\frac{\partial \boldsymbol{V}_i^c}{\partial x_i^s} = \sum_n^H \sum_m^W \boldsymbol{U}_{nm}^c \text{max}(0, 1 - |y_i^s - n|) \begin{cases} 0 & \text{if } |m - x_i^s| \geq 1 \\ 1 & \text{if } m \geq x_i^s \\ -1 & \text{if } m < x_i^s \end{cases}, \tag{2.31}$$

$$\frac{\partial \boldsymbol{V}_i^c}{\partial y_i^s} = \sum_n^H \sum_m^W \boldsymbol{U}_{nm}^c \text{max}(0, 1 - |x_i^s - m|) \begin{cases} 0 & \text{if } |n - y_i^s| \geq 1 \\ 1 & \text{if } n \geq y_i^s \\ -1 & \text{if } n < y_i^s \end{cases}. \tag{2.32}$$

2.6 Joint Alignment

Alignment. CNNs and STNs are used in not only classification tasks but also alignment tasks [20, 8, 9, 56]. For example, de Vos *et al*. [8] used STNs to align pairs of 2D images, using a similarity metric as the overall loss function. This was later adapted to 3D images [9]. Building upon STNs, Rocco *et al*. [34] also showed good alignment results when working with pairs of images. These approaches were developed only for alignment

tasks and focus on aligning only two images, namely a template image and an example image. The alignment process warps the example image in multiple steps until it is aligned with the template. However, the previous STN-based methods would not be useful in alignment of more than 2 images, since no template image is given for each batch during training in the classification task. One possible solution is to use the average image of the batch as the template, and align all images to it, in a pair-wise manner. However, using the average image may not produce desirable alignment results [6, 38], since the images may be too far apart, and thus the average image may not hold any useful information (see, for example, the fourth row of Fig. 4.1). Another option is randomly choosing an image from the batch as the template. This approach might be problematic when the randomly chosen template does not well represent the content of the image. For instance, in digits alignment, when choosing randomly the template of the digit '7', if the '7' resembles a '1', the overall classification results would suffer.

Huang *et al*. [20] used neural networks to jointly align a batch of imaging without a template image, but only used the network as a feature extractor and not as an integrated learning process. In our work, we propose an end-to-end learning approach that uses STN as an alignment mechanism for a group of input feature maps and demonstrate its value in both alignment and classification tasks. The alignment loss function is added to the classifying network's loss function, hence making alignment an integral part of the learning process.

Congealing. Congealing is a classic algorithm for the joint alignment of a set of images (more generally, signals) [24]. Typical alignment criteria used in congealing algorithms are entropy minimization [29, 24, 19, 28, 20] and least-squares [6, 7]. These methods focus on inferring the joint alignment of a collection of signals, and do not learn; namely, they do not offer a mechanism for jointly aligning new signals with each other. Particularly, in [20] the authors propose a two-step process. The first step involves learning the CNN features, and is unrelated to alignment. Next, and not as part of an end-to-end learning, the congealing algorithm is used to align these features.

2.7 Motivation for the Proposed Method

Alignment loss. The basic goal of STNs is to perform any possible transformation (given

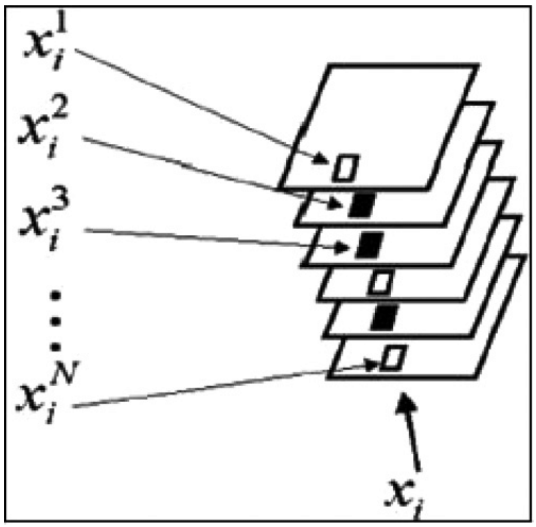


Figure 2.14: Congealing as a mechanism to perform joint alignment. At each iteration, the algorithm transforms the images in a way which minimizes a certain alignment criteria which is measured on each of the pixel stacks in the stack of images. Image Credit: [24]

the predefined transformation family) which can help reduce the learning loss. Since the loss function in deep learning can easily have millions of parameters and is usually not convex, then this function will have many local minimum points. Many of these points might be 'good' minimum points which are relatively close to the true global minimum, but likely there will also be a few 'bad' minimum points, *i.e.*, minimum points which are far from the global minimum and from which it is hard to escape from once trapped in during the training process. This implies that there may be many different types of transformations which can help the network arrive at a reasonably good minimum point, while there are some transformations which will get the network stuck in an undesirable location. Since in each step of SGD the network takes a step in a direction which pushes it closer to a relative minimum point (with respect to the current location), there may very well be many options of different transformations for pushing the network towards a local minimum during backpropagation. Because of this, in all likelihood the STN will not learn parameters which cause the input images to align with each other, even if alignment makes sense to us (in our human capacity) given the dataset, and even if alignment could help avoid the undesirable minimum points. The main achievement of our work is to demonstrate that pushing STNs to align the input feature maps via an alignment loss, can help classification in cases where alignment makes sense given the transformation family.

STN regularization. When learning spatial transformations there is a risk that the

Figure 2.15: Cluttered MNIST: Alignment as an auxiliary task to classification. Rows 1-3 are random images from the Cluttered MNIST dataset (here shown the 60-by-60 case; see Fig. 1.2 for the simpler case). Row 4 is the average of all test images per class, blurred due to misalignment. Row 5 is the average of the transformed test images using an STN-based classifier trained with the proposed alignment term as an auxiliary term. The alignment was learned jointly with the classification. The alignment term was measured on deep features and not on the transformed images. Thus, the relative blurriness in Row 5 is to be expected (as the upstream max-pooling layers prevented high accuracy of the alignment as they discarded small perturbation) though it is much sharper than Row 4.

learning will not converge to befitting transformations. Ensuring convergence into 'good' transformations via STNs is difficult, since it is hard to train neural networks with STNs, and is even more difficult when the networks are deep. However, adding a regularization term might help with avoiding undesirable transformations [24, 40]. In our work we have demonstrated that in order to achieve good classification results via STNs on real-world image datasets, adding a regularization term on the learned transformations is a crucial step in preventing the STN from converging into 'bad' transformations during the learning process. To the best of our knowledge, this is the first work that focuses on, and demonstrates the effect of using this regularization term in a thorough manner.

When using an STN-based network for classification purposes, we find that our alignment approach, using the regularization term, supports the STN and encourages it to learn the ('good') transformations needed for optimal classification. Our approach also prevents

over-fitting, as shown in the results section.

STN-based training in very deep networks. In very deep networks (such as Resnet, DenseNet), it is not obvious that STN's can significantly improve classification results, or will even have a positive effect at all, as we show in section 4.1 below. The reason why it is harder to train STNs in very deep nets than in smaller nets, is that in these very deep networks the network learns sophisticated features that capture the true class in a way that may already be partially invariant to spatial transformations. For these networks, the need to learn extra transformation parameters during the learning process seems to disturb the learning process, possibly making the regular STN redundant. Our work demonstrates that for datatsets of reduced sizes, adding a regularization term alongside the proposed alignment loss can still boost the performance of the very deep baseline classifiers.

Chapter 3

The Proposed Method

3.1 Method

Suppose we are given N training examples, $(x_i)_{i=1}^N$. Let $z_i \in \{1, 2, \ldots, K\}$ denote the label of x_i where K is the number of classes. The proposed method uses ST layers to execute two tasks simultaneously: the original classification task and the auxiliary task of within-class alignments; *i.e.*, for every $j \in \{1, 2, \ldots, K\}$, this second task targets the alignment of $\{x_i : z_i = j\}$. Thus, during training, the network learns to predict transformations that enable execution of both of the aforesaid tasks. Given a baseline network, it is modified by placing one or more ST layers directly after the input images (this, in itself, is not new [21]). To specify an ST layer, a specific transformation family is chosen. In our experiments, we used affine transformations. More generally, one can use more expressive transformations such as homographies [26], thin-plate splines [21], diffeomorphisms [47] or beamforming [49].

3.1.1 The Alignment Loss Term

We will later explain how we choose which feature should be aligned. Let F denote that chosen feature, obtained from x_i. For example, F can be taken to be the output of the ST layer(s), in which case, since we placed the ST layers at the network's beginning, right after the input images, F will be $\boldsymbol{V}$, the transformed image. Another possibility is letting F be a feature from a later stage of the network; *e.g.*, the output of one of the conv layers. The alignment loss function, used to align an ensemble of feature maps, is based on within-class least-squares criterion (similar to, *e.g.*, the criterion used in least-squares

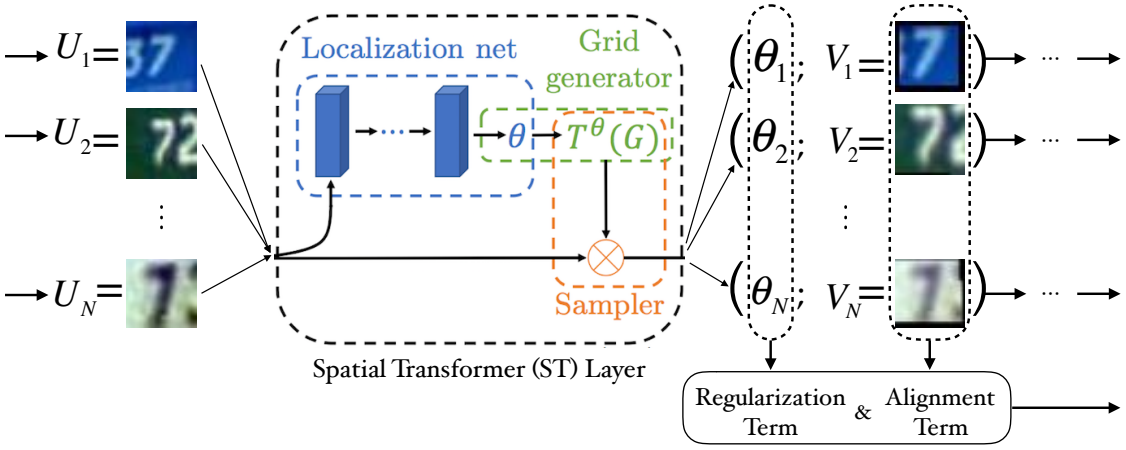


Figure 3.1: An example architecture of the proposed method. Here, an ST layer is placed after the input images (taken from the challenging, real-world SVHN dataset), and the alignment loss is measured directly on the transformed images (more generally, and not shown in this figure, we sometimes measure that loss on deeper features). The optional regularization term penalizes large transformations. This figure was adapted from [21] to account for the proposed alignment loss and the regularization loss.

congealing [6, 7]), as explained below.

Given D two-dimensional feature maps associated with a single image, where each map has H rows and W columns, the shape of the feature stack for that image is $[H, W, D]$; *e.g.*, if the feature maps represent the RGB values of a single 32-by-32 color image (transformed by an ST layer), then the shape is $[H = 32, W = 32, D = 3]$. The shape of the feature stack obtained from all N images is thus is $[N, H, W, D]$. Let

$$r \in \{1, \ldots, H\}\,,\; c \in \{1, \ldots, W\}\,,\; d \in \{1, \ldots, D\}\,, \tag{3.1}$$

let $F_{i,r,c,d}$ denote the value of the d-th feature map of the i-th image at location (r, c), and let

$$F_{r,c,d} = (F_{1,r,c,d}, \ldots, F_{N,r,c,d}) \in \mathbb{R}^N \tag{3.2}$$

denote the pixel stack of all the images at feature d at location r, c. Let $\widehat{\mu}_{r,c,d,j}$ and $\widehat{\sigma}^2_{r,c,d,j}$ denote the sample mean and sample variance of $\{F_{i,r,c,d} : z_i = j\}$ (*i.e.*, restricting the stack

to examples labeled as class j):

$$\widehat{\mu}_{r,c,d,j} = \frac{1}{n_j} \sum\nolimits_{i:z_i=j} F_{i,r,c,d} \tag{3.3}$$

$$\sigma^2_{r,c,d,j} = \frac{1}{n_j} \sum\nolimits_{i:z_i=j} (F_{i,r,c,d} - \widehat{\mu}_{r,c,d,j})^2 \tag{3.4}$$

where $\mathbb{1}_{z_i=j}(z_i)$ is 1 if $z_i = j$ and is 0 otherwise, and

$$n_j = \sum\nolimits_{i=1}^{N} \mathbb{1}_{z_i=j}(z_i) = |\{i : z_i = j\}| \tag{3.5}$$

is the number of examples labeled as class j. The overall within-class variance of the d-th feature map, for class j, is:

$$L^{\text{align}}_{d,j} = \frac{1}{H \times W} \sum_{r=1}^{H} \sum_{c=1}^{W} \sigma^2_{r,c,d,j} \,. \tag{3.6}$$

The robust, within-class variance for class j is then defined as:

$$L^{\text{align}}_{j} = \underset{d}{\text{median}} \left\{ L^{\text{align}}_{d,j} \right\}. \tag{3.7}$$

We use the sample median (as opposed to the mean) since usually not all learned feature maps represent features that should be aligned; *i.e.*, the latter can be viewed as outliers (or, in statistical lingo, nuisance parameters) w.r.t. the alignment task. We also empirically found that the robustness of the sample median lets it significantly outperform the sample mean, $\frac{1}{K} \sum_{d=1}^{K} L^{\text{align}}_{d,j}$. Finally, our overall alignment loss is given by summing L^{align}_{j} over all classes:

$$\boxed{L^{\text{align}} \triangleq \sum_{j=1}^{K} L^{\text{align}}_{j}} \,. \tag{3.8}$$

Shallow networks. When testing the proposed method on "shallow" networks (4 layers with 2 conv layers), it easily outperforms the baseline network by aligning the first or second conv feature maps. In these feature maps the general outline of the object being classified from the original image is still visible. Thus, when we align such feature maps, we obtain that the output of the ST layers is an aligned version of the input.

In the case in which we align the output of the second conv layer, the average image will still seem a bit blurry. This is expected since, in this case, we align feature maps which were effectively down-sampled via max-pooling, so the alignment does not take into account small translations (see Fig. 4.1 below for a practical example).

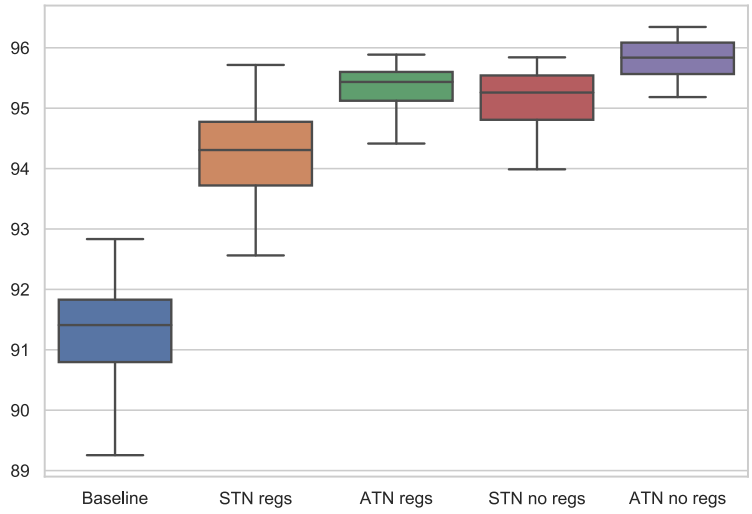


Figure 3.2: Box plot and error rates (%) for 50 experiments on the 60-by-60 Cluttered MNIST with a 4-layer CNN as the baseline. From left to right: 1) the baseline (7.17%); 2) the baseline + ST layers + a regularization term (4.28%); 3) same as #2 but with the proposed alignment term measured after the second conv layer (4.11%); 4) same as #2 but without the regularization (4.16%); 5) same as #3 but without the regularization (3.66%).

Very deep networks. As previously described, in very deep networks it is not obvious that STN's can significantly improve classification results, or will even have a positive effect at all, as we show in section 4.1 below. The reason why it is harder to train STNs in very deep networks than in smaller networks, is that in these very deep networks the network learns sophisticated features that capture the true class in a way which might already be partially invariant to spatial transformations. For these networks, the need to learn extra transformation parameters during the learning process seems to disturb the learning process, possibly making the regular STN redundant. Our work demonstrates that for datatsets of reduced sizes, adding a regularization term alongside the proposed alignment loss can boost the performance of the very deep baseline classifiers.

Choosing What to Align. An important design choice is on what feature maps to base the alignment; *i.e.*, at which point in the network we should measure the alignment loss.

One straightforward choice is to measure the alignment loss on the images right after they are transformed by the ST layer(s). A second choice, however, is to measure that loss on the output of one (or more) of the conv layers in the middle of the network. As

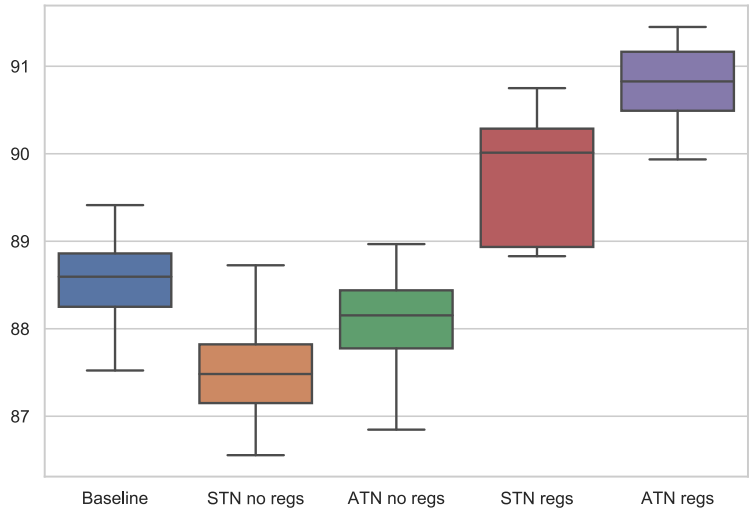


Figure 3.3: Box plot and error rates (%) for 50 experiments on SVHN with a 4-layer CNN as the baseline. From left to right: 1) the baseline CNN (10.59%). 2) the baseline + ST layers without a regularization term (11.92%). 3) as #2 but with the proposed alignment term calculated on the output of the second conv layer (11.03%). 4) Same as #2 + a regularization term (9.25%). 5) Same s #3 + a regularization term (8.55%).

demonstrated in section 4.1 below, in certain situations this second choice is preferable to the first one. A case example in which the second choice is preferable is that of datasets where the original images contain many outliers in random locations that are irrelevant to the class label. This situation is typified by the cluttered MNIST dataset. As shown by Zeiler *et al*. [55], the feature maps produced by the first conv layer usually represent edges and/or colors, while as we move further in the network we obtain higher-level representations. As a result, when we use deep features, many of the irrelevant and obfuscating details in the original images have already disappeared, making alignment easier.

3.1.2 Optional Regularization

The goal of the regularization term is to discourage too-large transformations. For example, transformations that either shrink the images to a point, or expand them such that one pixel of the original image fills the entire new image, will both cause the alignment term to vanish as the variance will decrease to zero. We used the regularization term proposed

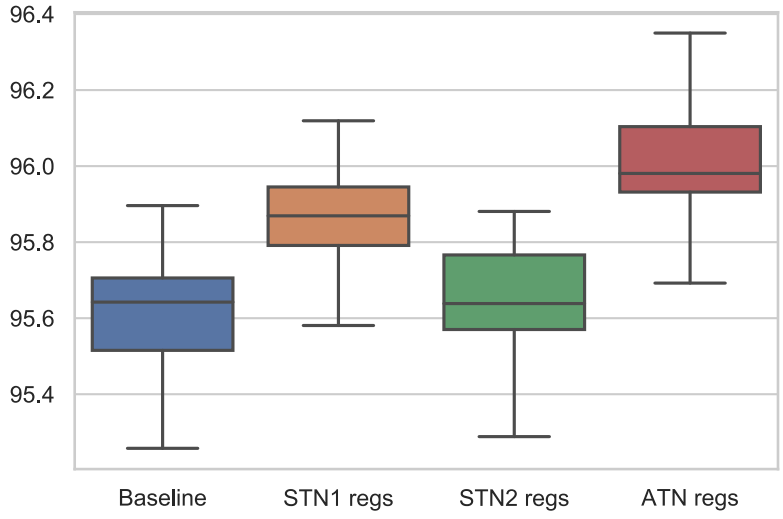


Figure 3.4: Box plot and error rates (%) for 50 experiments on SVHN with half the training data and DenseNet as a baseline. For all STN-based networks referenced here, we used a regularization term. 1) DenseNet (4.10%); 2) DenseNet + ST layers after the inpurt images (3.88%) 3) DenseNet + ST layers after deep feature maps (4.12%); 4) DenseNet + ST layers after input images and the proposed alignment term measured on the deep feature maps (3.65%).

in [24] (a non-DL framework for the alignment of multiple images), namely, the squared Frobenius distance between the average transformation matrix and the identity matrix. For example, for affine maps, the regularization term for the entire batch is

$$L^{\text{reg}} = \left\| \left(\frac{1}{N} \sum_{i=1}^{N} \boldsymbol{H}_i \right) - \boldsymbol{I}_{3\times 3} \right\|_F^2 \tag{3.9}$$

where the 3×3 matrix $\boldsymbol{H}_i$ stands for the predicted affine transformation for the i-th example. Since this regularization first calculates the average transformation, it is especially useful for alignment tasks, because, for example, if the transformation for one image is translated to the right, then for another image we expect the transformation to be translated to the left, so that, on average, no transformation is performed while alignment can be achieved.

Finally, the overall loss now has the form

$$L^{\text{classification}} + \lambda_1 L^{\text{align}} + \lambda_2 L^{\text{reg}} \tag{3.10}$$

where $L^{\text{classification}}$ is the original classification loss.

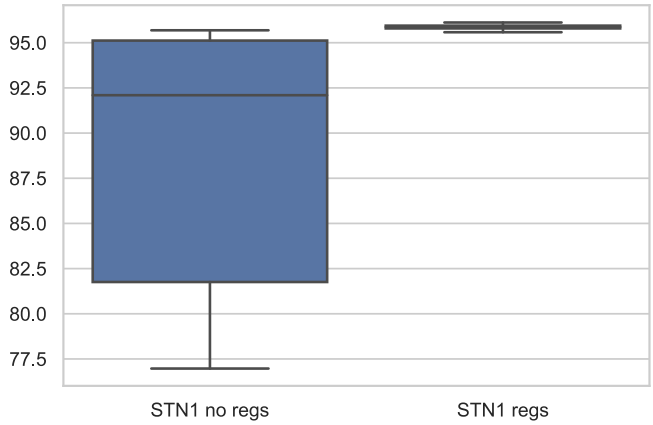


Figure 3.5: The importance of STN regularization in very deep networks. Box plot and error rates (%) for 50 experiments on SVHN with half the training data and DenseNet as a baseline. 1. DenseNet + ST layers after the input images without regularization. 2. DenseNet + ST layers after the input images with regularization.

3.1.3 Training

The combined loss is made of the global loss of the network's task at hand and the alignment loss (with or without the regularization term as well). In order to achieve the best results, we found that it is preferable during the training epochs to iterate between using the combined loss as the loss function and using only the original classification loss. This has the effect of letting the classifier learn its task, while still pushing it toward alignment. When we align deep features (*i.e.*, features learned by one of the internal conv layers), we let the networks learn for a few epochs before we add both the ST layers and the alignment loss into the learning process. This showed to be a good practice with STNs even when the proposed alignment loss is not used as it enables the network to first learn some basic, descriptive feature maps before trying to transform completely meaningless random states. This is why when we trained DenseNet [17] on SVHN with an STN (with or without the additional loss terms), we ran two more epochs than used in [17]. We found that in this manner even the baseline architecture (*i.e.*, just DenseNet) obtains slightly better results than reported in [17].

Supervision and Semi-supervision. The original classification task is supervised. The auxiliary alignment task is semi-supervised. During training, we use the known training labels to measure (and optimize) the within-class variances, but even during learning, we

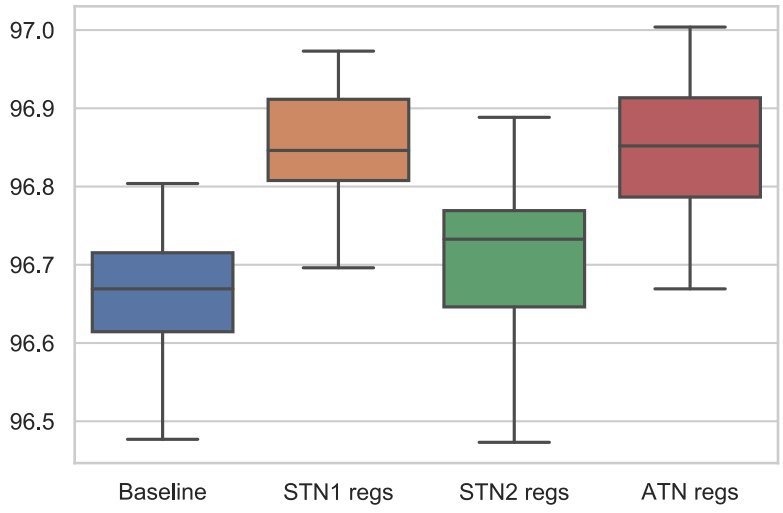


Figure 3.6: A limitation of the proposed method. Box plot and error rates (%) for 50 experiments on SVHN with all the training data and DenseNet as a baseline. The networks are the same as in Fig. 3.4. When using the entire SVHN training data with a state-of-the-art, very deep networks as baseline, the gain of the proposed method over the regular STN is less noticeable, but still outperforms the baseline network. When working with all training data and all with all the 'extra' data, we find that baseline DenseNet achieves superior results similar to those achieved by the STN-based methods, and thus for a dataset of such magnitude, the proposed method does not give an extra gain in performance over DenseNet.

neither use nor know the 'true' latent transformations associated with the training data (*i.e.*, the training data is misaligned and does not come with "training transformations"). During the test phase, neither the latent test labels nor the latent transformations are used.

Chapter 4

Results

4.1 Results Using The Method

We utilized the proposed method and tested the hypothesized loss terms on different CNN architectures and datasets. A description of our testing and the results produced are set out below.

4.1.1 Datasets

We experimented with two different versions of Cluttered MNIST [21], and with the more challenging, real-world dataset of Google's Street View House Numbers (SVHN) [31]. Cluttered MNIST was derived from the MNIST dataset [25], where each original 28-by-28 MNIST image was embedded at a random location within a larger image with random clutter. There are 50K training images and 10K test images. The two versions of Cluttered MNIST differ from each other in the size of the large image, and in the amount of clutter. The first consists of 40-by-40 images with little clutter noise (see, *e.g.*, Fig. 1.2), the second consists of 60-by-60 images with more noise (see, *e.g.*, Fig. 4.1). Moreover, due to the image size, the latent transformations in the first version are much smaller than in the second. Thus, the first version is considered easier than the second (though both are harder than vanilla MNIST).

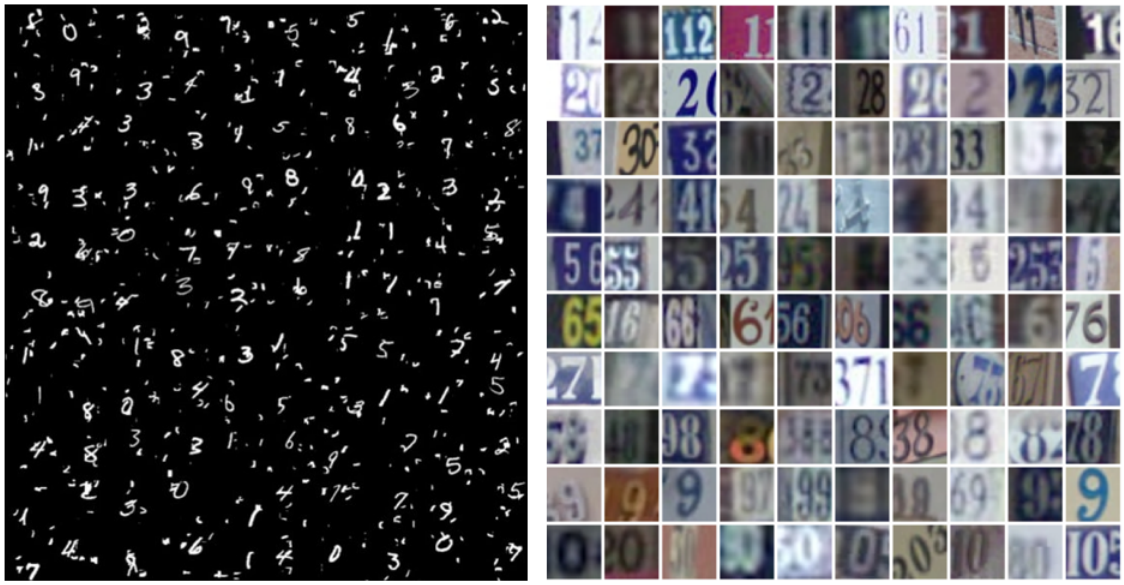


Figure 4.1: 60-by-60 Cluttered MNIST and SVHN datasets.

4.1.2 Baseline Architectures

As baselines, we experimented with both relatively-shallow and very-deep architectures. The relatively-shallow network was a simple 4-layer CNN that consisted of 2 conv layers followed by 2 FC layers. The deeper-net baseline was a DenseNet of 40 layers (and a growth factor of $k = 12$). In all experiments with STNs, the localization network was a 2-layer FC network in which the first hidden layer consisted of 20 neurons and the second consisted of $D = \dim(\boldsymbol{\theta})$ neurons, where $\boldsymbol{\theta}$ represents the parameters of the chosen transformation family: *e.g.*, 6 for affine maps.

4.1.3 Number of ST Layers

In our experiments, we used either 1 ST layer or 2 consecutive ST ones. We did not use more than 2 STNs, since the interpolation artifacts accumulate and hurt the results, especially in small images like the 32-by-32 images in SVHN.

4.1.4 General Experiment Design and Analysis

One of the problems with published research results in general, and specifically with published results obtained using deep learning, is that the writers often publish a single result which achieved a superior score, while the readers do not know how many times the experiment (*i.e.* training the network) was run until this score was obtained and what

the score was in those experiments which did not achieve the best result. This range in results is possible due to the fact that there is quite a lot of randomness occurring during the neural network training phase (*e.g.*, random initialization which enables the network to proceed to a local minimum starting at a completely random location).

Bearing this in mind, we trained 50 independent models (*i.e.*, 50 independent runs) for each architecture and dataset, and we then measured performance of each model in terms of classification accuracies on the test data. The boxplots we present, summarizing these comprehensive experiments, include results for: 1) a baseline architecture; 2) an STN = the baseline + ST layers; and 3) an Alignment Transformer Network (ATN), which is the STN together with the proposed alignment loss. Additionally, we show in these boxplots different regularization settings, depending on the experiment.

4.1.5 Cluttered MNIST: 40-by-40

Since in this dataset the challenge is only in learning small translations and in overcoming a little clutter, for this dataset we have shown that by using the proposed loss directly on the (transformed) input images, an ATN (based on the shallow baseline) was able to achieve good alignment; see Fig. 1.2. This experiment demonstrates that the proposed alignment loss can indeed be optimized successfully.

4.1.6 Cluttered MNIST: 60-by-60

As Fig. 4.1 shows, on this more challenging dataset the average images are completely blurred due to the wide distributions of the locations of the digits and the clutter noise. Aligning the (transformed) input images by minimizing their variance no longer makes sense as the task is too hard. Thus, here we chose to align the feature maps generated after the second conv layer and allow the network to learn the best features for classification. The baseline is again shallow. Since the noise is downplayed by such features, the alignment task becomes robust to it. At the same time, the alignment task helps the classification task improve over the baseline. Each of the 50 runs (of each experiment) consisted of 40 epochs. For the STN/ATN, we used a single affine ST layer. The results are summarized in Fig. 3.2. In this relatively-simple man-made dataset, there is less need for regularization. When adding the alignment loss, the ATN surpasses both the STN with or without regularization.

4.1.7 SVHN

This dataset consists of 32×32 pixel images centered around a single digit, while usually there are also additional nuisance digits on its side(s). The training set had 73,257 digits, and the test set 26,032 digits (there are also additional 531,131 extra training examples that are considered easier and less interesting; we did not use these). We tested in 3 different settings: **1. Baseline = Shallow CNN.** In this case, each of the 50 runs summarized in boxplots (Fig. 3.3) had 50 epochs where we trained on all 73,257 examples. As Fig. 3.3 shows, the benefits of the ATN as well as the fact that for this complicated dataset, adding regularization is important. **2. Baseline = DenseNet.** In this case, the proposed method proved especially effective in the small-data regime. The authors of the DenseNet paper [17] wrote that they "observe that dense connections have a regularizing effect, which reduces over-fitting on tasks with smaller training set size". The importance of creating models that achieve good results when using less data is useful in many applications. Our results show that on a reduced subset of the training data, the proposed alignment method produces superior results in comparison with the baseline DenseNet. We note, in particular, that we reduced the training data to 37K training images (about half the original quantity). This experiment thus tests the performance of a state-of-the-art classifier on reduced data and shows that it can still be improved when adding ST layers, especially when our proposed loss is added; see Fig. 3.4. Here each of the 50 runs corresponded to 42 epochs. We tested the same configurations as with the shallow baseline, with the following changes: 1. In the STN networks we used two ST layers. 2. As before, we tested the STN with and without regularization. The results of these two configurations are compared in a separate figure (Fig. 3.5), since the worst accuracy that was achieved by the network without any regularization was too low (76.9). For comparison, the worse score of the *baseline* was 95.25. Therefore, including the results of the no-regularization STN with the others, would make differentiating between the other networks impossible. 3. We aligned the feature maps generated by the first conv layer in the third Dense-Block (*i.e.*, aligning pretty deep features); 4. Since we were aligning deep features, we also tested the network with ST layers and no alignment loss, in which the ST layers appear straight after the feature maps which we chose for the alignment task. This shows that the alignment term has an additional positive effect on the learning process.

4.1.8 Limitations of the Proposed Method.

Very deep networks with 'enough' training data. When using the DenseNet as a baseline for SVHN classification, and working on the entire training data without the 'extra' data, we still outperformed the baseline DenseNet, but found that the performance gain over the regular STN with the regularization term was hardly noticeable. (see Fig. 3.6 above). when testing the proposed method on the entire training data with all of the 'extra' images, we found that the baseline DenseNet performed just as well as the STN based method and generated comparable results. We hypothesize that if we feed state-of-the-art, relatively deep networks with enough training images, two things will happen: 1. It will be easier for the regular STN to perform alignment and thus obtain similar results to the proposed method without an alignment term. 2. The baseline network, itself, will be able to learn features which capture the classes in such a way that is already invariant to spatial transformations, and therefore alignment will not affect the results. Note, that since the 'extra' dataset contains seven times more images than the regular training data, using this dataset can in itself be considered as a form of data augmentation.

"Unalignable" datasets. For the alignment approach to succeed, it is imperative that the chosen transformation family be expressive enough to enable it to well align the data. As a negative example, we tested the approach on the CIFAR-10 dataset using affine ST layers. Unlike the Cluttered MNIST or SVHN, here the great non-rigid pose variability within each class, together with the significant partial occlusions, implies that either transformation type is not rich enough to well align the data. Thus, we found that although we were able to bring down the alignment loss for such datasets, it had little effect on the classification results, and the ATNs were not able to improve upon either of the baselines (simple CNN and DenseNet).

Chapter 5

Conclusion

Our work demonstrates a method for improving STN-based classification by way of an auxiliary alignment loss. Via extensive experimentation and thorough analysis of the results, we have shown that on "alignable" datasets, the alignment term improves classification. In addition, we have explored and demonstrated the parameters for which it is beneficial to add a regularization term together with utilizing the alignment term. Prior to our work, regularization terms (the same or similar to those which we employed) had often been used, but had not been well studied, such that detailed data and in-depth results (as we have provided) were not available. The results that we have generated demonstrate the utility of applying the approach for simple networks to either artificial (Cluttered MNIST) or real-world (SVHN) datatests. Our results also show its utility when working with real-world data and a very deep network (DenseNet). In summary, training STN-based classifiers is often difficult; we believe our work is the first to demonstrate the importance of STN regularization during the learning process and the first to show that in some cases alignment can help the STN-based learning process. We hope that the analysis and results of our work will help make STN-based training more accessible to the research community in the future.

Appendix A

Set out below are additional, technical details needed for implementation of the proposed method.

A.1 Spatial Transformer (ST) Layers

As we previously described, when adding ST layers to baseline networks, we used either a single ST immediately after the input images or 2 consecutive ST layers (either immediately after the input images or after the deep feature maps). In all our experiments we worked with the following settings:

A.1.1 Localization Network

Whenever an STN was used, its localization network was a 2-layer Fully-Connected (FC) network in which the first hidden layer consisted of 20 neurons and the second consisted of $D = \dim(\boldsymbol{\theta})$ neurons, where $\boldsymbol{\theta}$ represents the parameters of the chosen transformation family: *e.g.*, 8 neurons for homographies or 6 for affine. For the first hidden layer we used a **tanh** activation function and a dropout of 0.2% during the learning stage.

A.1.2 Optimization

We found that when learning the STN parameters and regardless of the optimization method that was used, empirically we obtained better results when initializing the learning rate to a relatively-small number ($\leq 10^{-3}$). Since in some cases the network to which we would like to add the ST layers achieves baseline results with a learning rate higher than that (*e.g.*, in the DenseNet they initialize the learning rate to 0.1), we optimized the

baseline networks parameters and the STN parameters with different learning rates. As we previously described, we also found that it is better to alternate between minimizing the loss constructed solely of the the classification task (and possibly a regularization term) on the one hand, and minimizing the total loss of the classification task plus the alignment loss (possible with a regularization term) on the other hand.

A.2 Cluttered MNIST

In the experiments which were run on the Cluttered MNIST dataset, the baseline architecture which was used was a 4-layer CNN, constructed of 2 conv layers followed by 2 FC layers. A single ST layer was used and the alignment loss was calculated on the output of the second conv layer. We used a learning rate of 10^{-3} for the baseline parameters, a learning rate of 5×10^{-4} for the ST layer parameters, and Adam as the optimization method in the learning process with a batch size of 64.

A.2.1 Conv Layers

In both layers we used filters of size 5×5, stride=1 and 'same' padding. For the first conv layer we used 16 filters while for the second we used 36 filters. Following each conv layer we placed a max-pooling layer with stride=2.

A.2.2 FC Layers

The first FC layer contains 128 neurons, has a dropout of 0.2% and for which we used Relu as the activation function. The second layers has K neurons, where K is the number of classes in the classification task and on which we calculate the cross entropy loss.

A.3 SVHN with a "Shallow" CNN as Baseline

In the experiments in which we used a shallow CNN as a baseline for the SVHN classification task, we used a 4-layer CNN, constructed of 2 conv layers followed by 2 FC layers. A single ST layer was used and the alignment loss was calculated on the output of the second conv layer. We used a learning rate of 10^{-3} for the baseline parameters, a learning rate of

10^{-3} for the ST layer parameters, and Adam as the optimization method in the learning process. A batch size of 512 was used during training.

A.3.1 Conv Layers

In both layers we used filters of size 5×5, stride=1 and 'same' padding. For the first conv layer we used 32 filters and for the second we used 64 filters. Following each conv layer we placed a max-pooling layer with stride=2.

A.3.2 FC Layers

The first FC layer contains 256 neurons, has a dropout of 0.7% and for which we used Relu as the activation function. The second layers has K neurons, where K is the number of classes in the classification task and on which we calculate the cross entropy loss.

A.4 SVHN with DenseNet as Baseline

In the experiments in which we used DenseNet as baseline for the SVHN classification task, we used an architecture similar to the one described in [17] with a depth of 40 and $k = 12$. As we previously described, the only difference is that we ran an extra 2 epochs, so overall we ran 42 epochs: the first 22 epochs with a 0.1 learning rate, the next 10 with a 0.01 learning rate and the last 10 epochs with a 0.001 learning rate. Also, we begin learning the ST parameters after the second epoch. In the experiments where we added ST layers and the proposed alignment loss to the DenseNet, 2 consecutive ST layers were used, and alignment was performed on the output of the first conv layer in the third Dense-Block. As in the DenseNet paper, here too the optimization method used was the Nesterov momentum. The STN parameters are trained with an initial learning rate of 10^{-4}, and in each epoch where the global learning rate is decreased, the STN learning rate is decreased as well by half its current value.

Bibliography

[1] V. Badrinarayanan, A. Kendall, and R. Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39, 2016. 1

[2] X. Cao, P. Wang, C. Meng, X. Bai, G. Gong, M. Liu, and J. Qi. Region based cnn for foreign object debris detection on airfield pavement. *Sensors*, 18(3):737, 2018. 2, 35

[3] Y. Chauvin and D. E. Rumelhart. Backpropagation: theory, architectures, and applications. 1995. 13

[4] L.-C. Chen, Y. Yang, J. Wang, W. Xu, and A. L. Yuille. Attention to scale: Scale-aware semantic image segmentation. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3640–3649, 2016. 1

[5] K. L. Chung. On a stochastic approximation method. 1954. 11

[6] M. Cox, S. Sridharan, S. Lucey, and J. F. Cohn. Least squares congealing for unsupervised alignment of images. *2008 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, 2008. 38, 43

[7] M. Cox, S. Sridharan, S. Lucey, and J. F. Cohn. Least-squares congealing for large numbers of images. *2009 IEEE 12th International Conference on Computer Vision*, pages 1949–1956, 2009. 38, 43

[8] B. D. de Vos, F. F. Berendsen, M. A. Viergever, M. Staring, and I. Isgum. End-to-end unsupervised deformable image registration with a convolutional neural network. In *Deep Learning in Medical Image Analysis and Multimodal Learning for Clinical Decision Support*. Springer, 2017. 37

[9] K. A. J. Eppenhof, M. W. Lafarge, P. Moeskops, M. Veta, and J. P. W. Pluim. Deformable image registration using convolutional neural networks. In *Medical Imaging: Image Processing*, 2018. 37

[10] R. B. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 580–587, 2014. 1

[11] A. W. Harley. An interactive node-link visualization of convolutional neural networks. In *ISVC*, pages 867–877, 2015. 2

[12] S. Hauberg, O. Freifeld, A. B. L. Larsen, J. W. F. III, and L. K. Hansen. Dreaming more data: Class-dependent distributions over diffeomorphisms for learned data augmentation. In *Proceedings of the 19th international Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 51, pages 342–350, 2016. 2

[13] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016. 1, 25

[14] J. F. Henriques and A. Vedaldi. Warped convolutions: Efficient invariance to spatial transformations. In *ICML*, 2017. 1

[15] A. G. Howard. Some improvements on deep convolutional neural network based image classification. *arXiv preprint arXiv:1312.5402*, 2013. 7

[16] J. Hu, L. Shen, and G. Sun. Squeeze-and-excitation networks. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7132–7141, 2018. 1

[17] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger. Densely connected convolutional networks. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2261–2269, 2017. 1, 2, 25, 48, 53, 58

[18] G. Huang, Y. Sun, Z. Liu, D. Sedra, and K. Q. Weinberger. Deep networks with stochastic depth. In *ECCV*, 2016. 25

[19] G. B. Huang, V. Jain, and E. Learned-Miller. Unsupervised joint alignment of complex images. In *ICCV*, pages 1–8. IEEE, 2007. 38

[20] G. B. Huang, M. A. Mattar, H. Lee, and E. G. Learned-Miller. Learning to align from scratch. In *NIPS*, 2012. 37, 38

[21] M. Jaderberg, K. Simonyan, A. Zisserman, and K. Kavukcuoglu. Spatial transformer networks. In *NIPS*, pages 2017–2025, 2015. 2, 35, 42, 43, 50

[22] J. Kiefer and J. Wolfowitz. Stochastic estimation of the maximum of a regression function. 1952. 11

[23] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, 2012. 1, 7, 23

[24] E. G. Learned-Miller. Data driven image models through continuous joint alignment. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 236–250, 2006. 38, 39, 40, 47

[25] Y. LeCun and C. Cortes. MNIST handwritten digit database. 2010. 1, 50

[26] C.-H. Lin, E. Yumer, O. Wang, E. Shechtman, and S. Lucey. St-gan: Spatial transformer generative adversarial networks for image compositing. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 9455–9464, 2018. 2, 42

[27] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. E. Reed, C.-Y. Fu, and A. C. Berg. Ssd: Single shot multibox detector. In *ECCV*, 2016. 1

[28] M. A. Mattar, M. G. Ross, and E. G. Learned-Miller. Nonparametric curve alignment. In *ICASSP*, pages 3457–3460. IEEE, 2009. 38

[29] E. G. Miller, N. E. Matsakis, and P. A. Viola. Learning from one example through shared densities on transforms. In *CVPR*, volume 1, pages 464–471. IEEE, 2000. 38

[30] M. Mirmohammadsadeghi, S. S. Hanna, and D. Cabric. Modulation classification using convolutional neural networks and spatial transformer networks. In *2017 51st Asilomar Conference on Signals, Systems, and Computers*, pages 936–939. IEEE, 2017. 2, 35

[31] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS Workshop*, 2011. 50

[32] G. Papandreou, L.-C. Chen, K. P. Murphy, and A. L. Yuille. Weakly-and semi-supervised learning of a deep convolutional network for semantic image segmentation. *2015 IEEE International Conference on Computer Vision*, pages 1742–1750, 2015. 1

[33] A. J. Ratner, H. R. Ehrenberg, Z. Hussain, J. A. Dunnmon, and C. Ré. Learning to compose domain-specific transformations for data augmentation. 2017. 2

[34] I. Rocco, R. Arandjelovic, and J. Sivic. Convolutional neural network architecture for geometric matching. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 39–48, 2017. 37

[35] O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation. In *MICCAI*, 2015. 1

[36] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986. 13

[37] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015. 7

[38] H. Schweitzer. Optimal eigenfeature selection by optimal image registration. In *CVPR*, 1999. 38

[39] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. *CoRR*, 2014. 1

[40] S. Shan, X. Guo, W. Yan, E. I.-C. Chang, Y. Fan, and Y. Xu. Unsupervised end-to-end learning for deformable medical image registration. *CoRR*, 2017. 40

[41] E. Shelhamer, J. Long, and T. Darrell. Fully convolutional networks for semantic segmentation. *2015 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3431–3440, 2015. 1

[42] A. Shrivastava and A. Gupta. Contextual priming and feedback for faster r-cnn. In *ECCV*, 2016. 1

[43] A. Shrivastava, A. Gupta, and R. B. Girshick. Training region-based object detectors with online hard example mining. *2016 IEEE Conference on Computer Vision and Pattern Recognition*, pages 761–769, 2016. 1

[44] P. Simard, B. Victorri, Y. LeCun, and J. S. Denker. Tangent prop – a formalism for specifying selected invariances in an adaptive network. In *NIPS*, 1992. 2

[45] P. Y. Simard, D. Steinkraus, and J. C. Platt. Best practices for convolutional neural networks applied to visual document analysis. In *ICDAR*. IEEE Computer Society, 2003. 2

[46] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, 2015. 1, 23

[47] N. Skafte Detlefsen, O. Freifeld, and S. Hauberg. Deep diffeomorphic transformer networks. In *CVPR*, 2018. 2, 42

[48] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015. 1, 23

[49] S. Vedula, O. Senouf, G. Zurakhov, A. Bronstein, O. Michailovich, and M. Zibulevsky. Learning beamforming in ultrasound imaging. *arXiv preprint arXiv:1812.08043*, 2018. 2, 42

[50] A. Veit, M. J. Wilber, and S. J. Belongie. Residual networks behave like ensembles of relatively shallow networks. In *NIPS*, 2016. 25

[51] X. Wang, A. Shrivastava, and A. Gupta. A-fast-rcnn: Hard positive generation via adversary for object detection. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3039–3048, 2017. 1

[52] D. E. Worrall, S. J. Garbin, D. Turmukhambetov, and G. J. Brostow. Harmonic networks: Deep translation and rotation equivariance. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 7168–7177, 2017. 1

[53] S. Xie, R. B. Girshick, P. Dollár, Z. Tu, and K. He. Aggregated residual transformations for deep neural networks. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5987–5995, 2017. 25

[54] R. Yamashita, M. Nishio, R. K. G. Do, and K. Togashi. Convolutional neural networks: an overview and application in radiology. *Insights into Imaging*, 9(4):611–629, 2018. 7

[55] M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. In *ECCV*, 2014. 22, 23, 46

[56] Z. Zhang, P. Luo, C. C. Loy, and X. Tang. Facial landmark detection by deep multi-task learning. In *European conference on computer vision*, pages 94–108. Springer, 2014. 37

אוניברסיטת בן־גוריון בנגב

חיבור לשם קבלת התואר "מגיסטר" בפקולטה למדעי הטבע

אשר פרדמן

ספטמבר 2019

ניתוח ההשפעה של רגיסטרציה ורגולריזציה על מסווגים מבוססי STN

תקציר

רשתות קונבולוציה עמוקות הניבו שיפורים מרחיקי לכת בתחום הראייה החישובית בכלל, ובפרט בבעיית סיווג אובייקטים (קלסיפיקציה). למרות זאת, רשתות אלו מוגבלות בכך שהן מושפעות מהמנח הגיאומטרי בו מצויים האובייקטים הנדרשים לסיווג. דהיינו, במידה ובוצעו על אותו אובייקט טרנספורמציות מרחביות שונות (כגון הזזה, סיבוב, הגדלה וכדו') רשת הקונבולוציה עלולה לטעות ולסווג את אותו אובייקט באופנים שונים. גישה נפוצה לפתרון בעיה זו היא להוסיף לרשת המקורית שכבה (אחת או יותר) שתפקידה ללמוד טרנספורמציות מרחביות ולהפעיל אותם על הקלט. באופן זה, הרשת המקורית אמורה להתעלם מהשינויים המרחביים ולהתמקד במאפיינים החשובים של כל אובייקט, כך שיתקבלו תוצאות סיווג אופטימליות. שכבה מודולרית זו נקראת STN. הניסיון מראה שבפועל קשה לאמן רשתות המכילות STN, במיוחד כאשר הרשת המקורית הינה רשת מאוד עמוקה כגון ResNet או DenseNet. על מנת לשפר את תהליך האימון של מסווגים מבוססי STN, אנו חוקרים בעבודה זו את ההשפעה של שתי התאמות למסווג: הוספת גורם רגולריזציה והוספת גורם הדוחף את STN ליישר את התמונות אלו עם אלו (alignment). שימוש ברגולריזציה עבור רשתות המכילות STN היא אמנם טכניקה המצויה בשימוש, אך ההשפעה של הוספת הרגולריזציה טרם נחקרה לעומק. בעבודה זו אנו מראים שבמקרים מסוימים, אי־שימוש ברגולריזציה גורם לSTN ללמוד טרנספורמציות הרסניות לתהליך הלמידה. התוספת של יישור התמונות היא חדשה בכל הקשור לשיפור מסווגים מבוססי STN. בדקנו את שתי ההתאמות הללו על רשתות שונות ועבור ערכות־נתונים (datasets) שונים. הניסויים שלנו מראים שעבור תמונות קלט הניתנות ליישור, שימוש בהתאמות אלו מביא לשיפור תוצאות הסיווג גם ברשתות מאוד עמוקות.

אוניברסיטת בן־גוריון בנגב

הפקולטה למדעי הטבע

המחלקה למדעי המחשב

ניתוח ההשפעה של רגיסטרציה ורגולריזציה על מסווגים מבוססי STN

חיבור לשם קבלת התואר "מגיסטר" בפקולטה למדעי הטבע

אשר פרדמן

תחת הנחייתו של:
ד"ר אורן פרייפלד

אלול תשע"ט ספטמבר 2019