Joint Spatial and Layer Attention for Convolutional Networks

by

Tony Joseph

A thesis submitted to the
School of Graduate and Postdoctoral Studies in partial
fulfillment of the requirements for the degree of

Master of Science in Computer Science

Faculty of Science
University of Ontario Institute of Technology.
Oshawa, Ontario, Canada
May 2019

This page is intentionally left blank.

Thesis Examination Information

Submitted by: **Tony Joseph**

Master of Science in **Computer Science**

Thesis Title:
Joint Spatial and Layer Attention for Convolutional Networks.

An oral defense of this thesis took place on *May 30, 2019* in front of the following examining committee:

Examining Committee:

Chair of Examining Committee	**Dr. Shahram Heydari**
Research Supervisor	**Dr. Faisal Qureshi**
Research Co-supervisor	**Dr. Kosta Derpanis**, *Ryerson University*
Examining Committee Member	**Dr. Ken Pu**
Thesis Examiner	**Dr. Mark Green**, *University of Ontario Institute of Technology*

The above committee determined that the thesis is acceptable in form and content and that a satisfactory knowledge of the field covered by the thesis was demonstrated by the candidate during an oral examination. A signed copy of the Certificate of Approval is available from the School of Graduate and Postdoctoral Studies.

Abstract

Joint Spatial and Layer Attention for Convolutional Networks

Tony Joseph

Faculty of Science (Computer Science)

University of Ontario Institute of Technology.

2019

In this work, we propose a novel approach that learns to sequentially attend to different Convolutional Neural Networks (CNN) layers (i.e., "what" feature abstraction to attend to) and different spatial locations of the selected feature map (i.e., "where") to perform the task at hand. Specifically, at each Recurrent Neural Network (RNN) step, both a CNN layer and localized spatial region within it are selected for further processing. We demonstrate the effectiveness of this approach on two computer vision tasks: (i) image-based six degree of freedom camera pose regression and (ii) indoor scene classification. Empirically, we show that combining the "what" and "where" aspects of attention improves network performance on both tasks. We evaluate our method on standard benchmarks for camera localization (Cambridge, 7-Scenes, and TUM-LSI) and for scene classification (MIT-67 Indoor Scenes). For camera localization our approach reduces the median error by 18.8% for position and 8.2% for orientation (averaged over all scenes), and for scene classification it improves the mean accuracy by 3.4% over previous methods.

Keywords: Computational Attention; Convolutional Neural Networks; Reccurent Neural Networks; Neural Networks; Image-Based Camera Localization

Author's Declaration

I hereby declare that this thesis consists of original work of which I have authored. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners. I authorize the *University of Ontario Institute of Technology* to lend this thesis to other institutions or individuals for the purpose of scholarly research. I further authorize the *University of Ontario Institute of Technology* to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research. I understand that my thesis will be made electronically available to the public.

Tony Joseph

(author)

Statement Of Contributions

Part of the work described in Chapter **4** and Chapter **5** has been published as:

> T. Joseph, K. Derpanis, and F. Qureshi. (2019) "Joint spatial and layer attention for convolutional networks", *arXiv preprint*, [Online]. Available: https://arxiv.org/abs/1901.05376.

I was responsible for performing all the experiments and writing of the manuscript. I have used standard referencing practices to acknowledge ideas, research techniques, or other materials that belong to others.

Acknowledgments

First I thank my advisor Faisal Qureshi who gave me this incredible opportunity to be part of his lab as a masters student. He always encouraged me to pursue what I was interested in, never held me back. He introduced me to Konstantinos G. Derpanis (my co-advisor) who further pushed me to do great work, as well as making me part of his lab. Their constant guidance and support were very crucial in completing this work. They transformed both my thinking and research abilities, making me more ambitious to pursue even harder problems.

I would like to thank my parents Joseph Kuriakose and Jessyamma Joseph, who are always very supportive and have sacrificed a lot to move our family to Canada. They taught me the value of hard work and to be fearless to pursue my dreams. And my close friends Murtaza Jawed and Mohammed Baneeh for encouraging and supporting me through hard times. I would also like to thank my close collaborator Kamyar Nazeri (Imaging lab, University of Ontario Institute of Technology) for providing support on figures presented in this work and for having intellectual discussions.

Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

Since 2012, deep learning based approaches have seen unprecedented success in many computer vision tasks, such as object detection [4], semantic segmentation [5], video tracking [6], motion estimation [7], image generation [8], etc. Convolutional Neural Networks (CNNs) [9] are central models in a broad range of computer vision tasks, e.g., [4,5,7,8,10]. Generally, the processing of input imagery consists of a series of convolutional layers interwoven with non-linearities (and possibly down-sampling) that yield a hierarchical image representation. The image representation constructed by CNNs are sometimes called *deep features*. These deep features, to a large extent, have displaced the hand-crafted features of old, which pre-date the wide-spread use of deep learning in computer vision by at least two decades.

As deterministic processing proceeds in a CNN, both the spatial scope (i.e., the effective receptive field) and the level of feature abstraction [11,12] of the representation gradually increase. Motivated by our understanding of human visual processing [13,14] and initial success in natural language processing [15], an emerging thread in computer vision research consists of augmenting CNNs with an atten-

tional mechanism. Generally speaking, the goal of attention is to dynamically focus computational resources on the most salient features of the input image as dictated by the task.

Deep features, especially those that are constructed by convolutional layers, encode spatial information, i.e, a given deep feature layer that picks a cat will also encode the location of this cat. When one is searching for a cat, knowing "where" to look can help. [16] developed an attention mechanism that focuses on different locations in the deep feature (or one might say that it selects features within the feature map that forms the deep feature) over successive steps. This attention mechanism, which we henceforth refer to as *soft attention*, has been applied successfully to the problem of image captioning [16].

It is straightforward to designate the image representations at a particular CNN layer as the deep feature that will be used during subsequent processing. It is also possible to combine image representations at multiple convolutional layers to construct the deep feature. It has been observed that different CNN layers capture information at different levels of abstraction [12]. Layers that are closer to the input capture fine-grained spatially localized structures, say edges and blobs; whereas, layers that are further away from the input capture more abstract information, such as existence of a cat or a dog. To the best of our knowledge, the decision about which CNN layer(s) provide the deep features is made at design time.

In this work, we present an approach that incorporates attention into a standard CNN in two ways: (i) a layer attention mechanism (i.e., "what" layer to consider) selects a CNN layer, and (ii) a spatial attention mechanism selects a spatial region within the selected layer (i.e., "where") for subsequent processing. Layer and spatial attention work in conjunction with a Recurrent Neural Network (RNN).

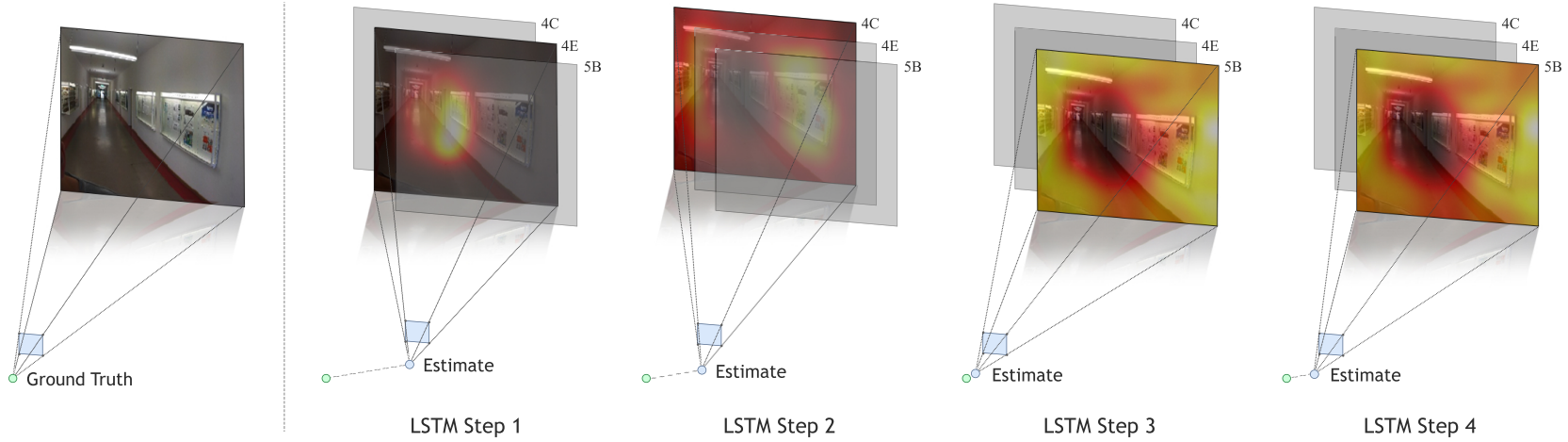

Figure 1.1: Overview of our approach to 6-DoF camera localization. Given a set of CNN feature layers (GoogLeNet [3] Conv-{3B, 4C, 4E, 5B} layers shown) our approach to attention uses an RNN to sequentially select a set of feature layers (highlighted by the non-grey images) and corresponding locations in the layers (highlighted by the heat maps). Finally, the processed attended features are used for regressing the camera position and orientation.

At each time step, first a layer is selected and next spatial attention is applied to it. The RNN progressively aggregates the information from the attended spatial locations in the selected layers. The aggregated information is subsequently used for regression or classification. Our model is trained end-to-end, without requiring additional supervisory labels. Empirically, we consider both regression (i.e., six degree of freedom, 6-DoF, camera localization) and classification (i.e., scene classification) tasks. Figure 1.1 presents an overview of our approach to layer-spatial attention for 6-DoF camera localization.

1.1 Contributions

This work makes the following contributions:

I. We propose an attention model that learns to sequentially attend to different CNN layers (i.e., different levels of abstraction) and different spatial locations (i.e., specific regions within the selected feature map) to perform the task at hand.

II. We augment a standard CNN architecture, GoogLeNet [3], with our attention model and empirically demonstrate its efficacy on both regression and classification tasks: 6-DoF camera localization regression and indoor scene classification. We evaluate the proposed architecture on standard benchmarks: (a) Cambridge Landmarks, 7 Scenes, and TU Munich Large-Scale Indoor (TUM-LSI) for camera pose estimation; and (b) MIT-67 Indoor Scenes for scene classification. For camera localization our approach reduced the overall median error by 12.3% for position and 13.9% for orientation on Cambridge Landmarks, 19.3% for position and 8.83% for orientation on 7-Scenes, and 25.1% for position and 1.79% for orientation on TUM-LSI over the baseline [2]. For indoor scene classification on MIT-67 [17] our approach improves the mean

accuracy by 3.4% over the baseline [18]. In both tasks, the baseline methods use the *same* base convolutional network.

1.2 Thesis Outline

Rest of the thesis is organized as follows. Chapter **2** provides some background on deep learning specifically CNN's, LSTM networks, and backpropagation for training these networks. Chapter **3** covers the literature review on the existing works in attention, camera pose estimation and indoor scene localization. Chapter **4** describes our methodology. Chapter **5** provides an overview of the experimental setup along with datasets used in this work along with both the quantitative and qualitative results. It also provides empirical motivation through an ablation study. Chapter **6** provides a conclusion and some potential suggestions for future work.

CHAPTER 2

Background

This chapter presents an overview of CNNs and LSTMs. For a more detailed discussion on mentioned concepts we recommend *the deep learning* book from Goodfellow *et al*. [19].

2.1 Neural Networks

Neural networks can be considered as functions that map an input space $\mathbf{X}$ to another space $\mathbf{Y}$, i.e. in the task of pose estimation, $\mathbf{X}$ could be the space of input images and $\mathbf{Y}$ represents the camera pose $[\mathbf{x},\ \mathbf{q}]^\top$, here $\mathbf{x} \in \mathbb{R}^3$ represents 3-D camera position and $\mathbf{q} \in \mathbb{R}^4$ camera orientation.

Neural networks are composed of individual *neurons* stacked together horizontally or vertically. Each neuron is a basic computational unit that is loosely based on biological neurons. An artificial neuron performs a weighted sum of the inputs $\mathbf{x}$, followed by applying a non-linearity σ. The implementation of a neuron is for-

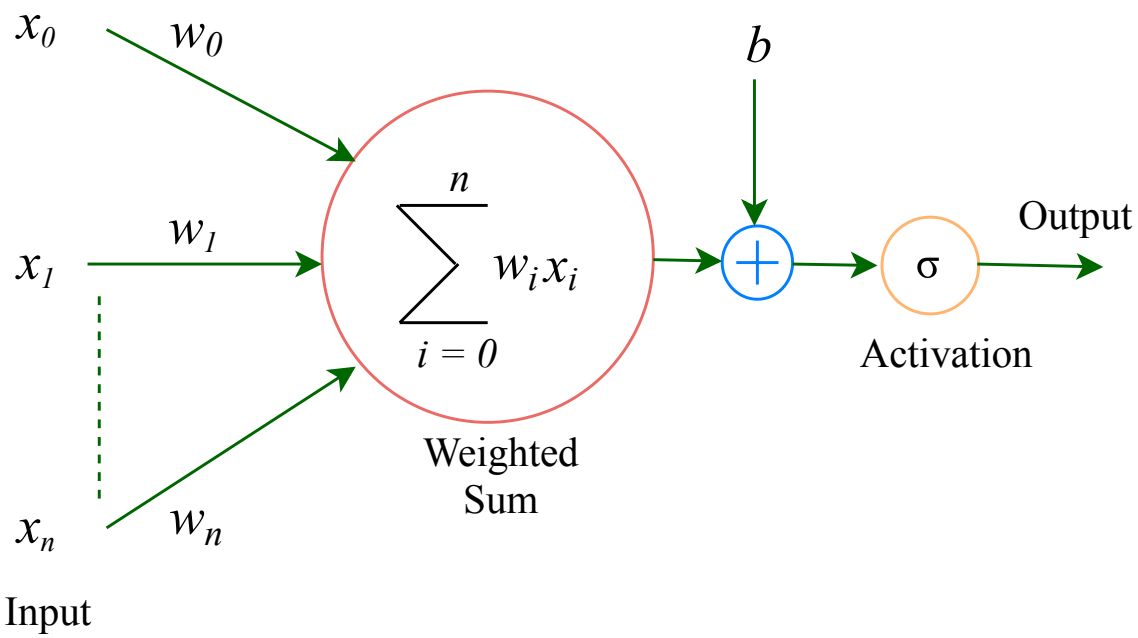

Figure 2.1: Block diagram illustrating an artificial neuron described in Equation 2.1. An input $x \in \mathbb{R}^{n+1}$. It then computes a weighted sum of the inputs, followed by applying an element-wise non-linearity, σ.

mulated as follows:

$$n(\mathbf{x}) = \sigma(\mathbf{w}^\top \mathbf{x} + \mathbf{b}) \tag{2.1}$$

where, $\mathbf{w}$ is the neuron *weights* and $\mathbf{b}$ is the *bias* term. Figure 2.1 shows an illustration of a neuron.

Further a network *layer* is formed by stacking m such neurons (m is a *hyperparameter*). The weights $\mathbf{w}$ from each m neuron together form a weight matrix, $\mathbf{W}$. The weighted sum vector of the layer is followed by applying an element-wise non-linearity, σ. Standard σ's used are sigmoid $\left(\frac{1}{1+e^{(-n)}}\right)$, tanh $\left(\frac{e^n - e^{-n}}{e^n + e^{-n}}\right)$, and ReLU $(max(0, n))$. By stacking such layers in a cascade setting, we get a *feed-forward multi-layer* neural network. An illustration of a two layer neural network model is shown in Figure 2.2. A two layered network is implemented as follows:

$$l(\mathbf{x}) = \mathbf{W}_2 \sigma(\mathbf{W}_1^\top \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2 \tag{2.2}$$

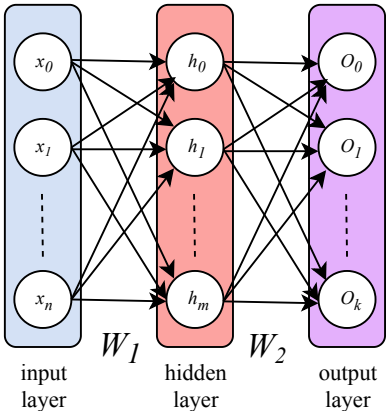

Figure 2.2: An example of two layer feed-forward multi-layer neural network as described in Equation 2.2. An input $\mathbf{x} \in \mathbb{R}^n$, which gets transformed to hidden layer using weight matrix $\mathbf{W}_1 \in \mathbb{R}^{m \times n}$ and into output by $\mathbf{W}_2 \in \mathbb{R}^{k \times m}$.

where $[\mathbf{W}_1, \mathbf{b}_1]$ is the weights and bias of layer-1. Similarly, $[\mathbf{W}_2, \mathbf{b}_2]$, is weights and bias of layer-2. Note that the final layer (output layer), in this case, layer-2 does not contain the non-linearity.

Summary and implementation details. We have discussed how to build neural network, using fully connected layers. When implementing a fully connected layer there is only one hyperparameter that is of concern: number of neurons m in the layer. Now let's show a general formulation of a fully connected layer:

I. **Input.** The input to the fully connected layer is a vector of size: x_i

II. **Parameter Estimation.** Since we have m neurons in a layer, the total learnable parameters are: $\mathbf{x}_i \,.\, m + 1$ (include bias).

III. **Output.** The output of a fully connected layer is a vector of size: m.

2.1.1 Convolutional Neural Networks

Convolutional Neural Networks (CNN) are neural networks that use convolutions instead of a general matrix multiply [19]. Compared to the standard neural network (fully-connected) which operates on vector inputs, $\mathbf{x} \in \mathbb{R}^d$, CNN can operate

on multi-dimensional inputs. This makes CNN architecture able to handle high dimensional data such as images, videos, etc. A color image $\mathbf{I} \in \mathbb{R}^{h \times w \times c}$ is a tensor[1] of height h, width w and channel c. For example let, $\mathbf{I} \in \mathbb{R}^{224 \times 224 \times 3}$ be a RGB image. When using a fully-connected neural network, the image $\mathbf{I}$, has to be converted to a vector, $\mathbf{I}_{vec} \in \mathbb{R}^{150528}$ (flatten the $\mathbf{I}$ tensor). This input vector $\mathbf{I}_{vec}$ has high dimensionality, which in turn requires a large number of network parameters and increases the processing time. Assuming layer-1 has 100 neurons, the total weights in layer-1 alone will be approximately 1.50×10^7.

In practice, this can become computationally intractable due to memory constraints. Such constraints practically hinders in building deep neural networks. Also, having such large parameters can result in network over-fitting on the training data and result in poor generalization, which is explained more in Section 2.2. This is where CNNs shines, with its ability to preserve spatial information and reduce network parameters through weight sharing.

Convolution. Assuming we have a 1-D input $\mathbf{I}^{'}$ and 1-D filter $\mathbf{k}$, the convolution operation $(*)$ is defined as:

$$(\mathbf{I}^{'} * \mathbf{k})[n] = \sum_{i=-\infty}^{+\infty} \mathbf{I}^{'}[n]\mathbf{k}[n-i] \tag{2.3}$$

note that in CNN convolution is implemented as shown in Equation 2.4, which is correlation. This operation extends to 2-D input as well, which is used in CNN. In deep learning context, it is referred to as *"convolution"*, because the filter weights are learned and if needed the network can learn to flip the filter weights, thereby resulting in convolution operation. Therefore, when we say *"convolution"* in CNN,

[1] tensors are generalizations of matrices with arbitary number of indices. Each index of a tensor can range over arbitary number of dimensions.

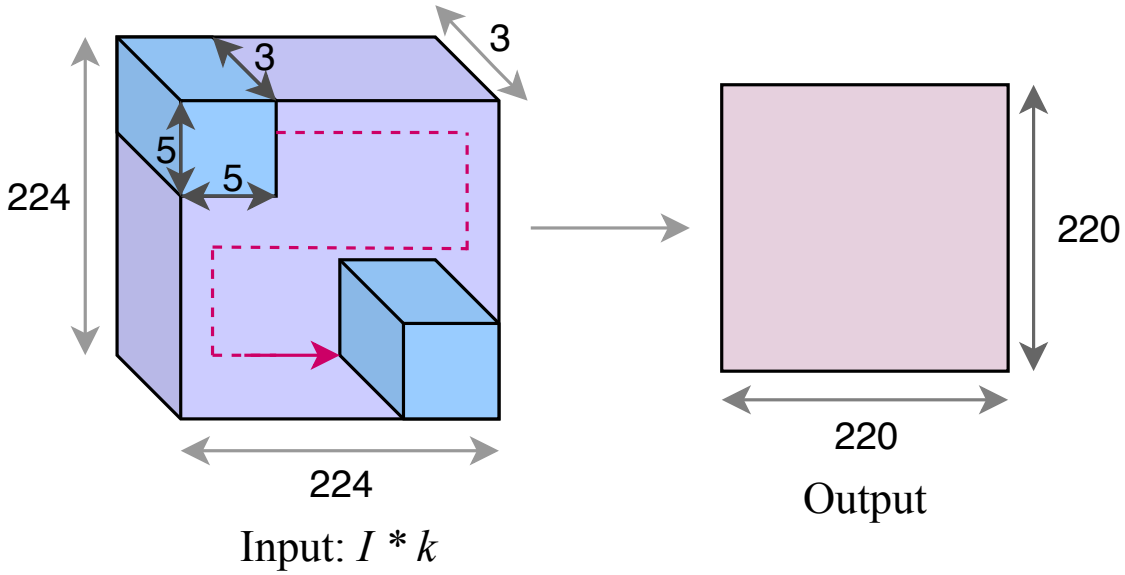

Figure 2.3: Block diagram illustrating a convolution on Image $\mathbf{I} \in \mathbb{R}^{224\times224\times3}$ with kernel $\mathbf{k} \in \mathbb{R}^{5\times5\times3}$ (padding $(P) = 0$, stride $(s) = 1$). Based on Equation 2.5, the resulting output tensor is $\mathbb{R}^{220\times220}$.

we are using Equation 2.4.

$$(\mathbf{I}^{'} * \mathbf{k})[n] = \sum_{i=-\infty}^{+\infty} \mathbf{I}^{'}[n]\mathbf{k}[n+i] \tag{2.4}$$

Convolutional Layer. The core building block of CNN is the Convolutional Layer. It takes in a tensor as input and outputs a tensor by convolving the inputs with a set of filters. This is well explained through an example. Let the input tensor be RGB image $\mathbf{I} \in \mathbb{R}^{224\times224\times3}$, which is to be convolved with a filter $\mathbf{k} \in \mathbb{R}^{5\times5\times3}$ (also referred to as *kernel*). The filter $\mathbf{k}$ has total of 75 learnable parameters $(5 \,.\, 5 \,.\, 3)$. We perform convolution by sliding the kernel across all spatial positions of the input image. At each position we compute a dot product between block (5 by 5 by 3) of $\mathbf{I}$ and $\mathbf{k}$. This will result in producing an output tensor (typically called an *activation map*) of $\mathbf{o} \in \mathbb{R}^{220\times220}$. An illustration of the convolutional layer operation is shown in Figure 2.3. Note that this is generalizable to any 3-D tensor.

Often, a convolutional layer will have multiple $\mathbf{k}$ such filters. Working with the same example of RGB image $\mathbf{I} \in \mathbb{R}^{224\times224\times3}$, which is to be convolved with m such $\mathbf{k} \in \mathbb{R}^{5\times5\times3}$ filters. The resulting output is a tensor of $\mathbf{o} \in \mathbb{R}^{220\times220\times m}$. Assuming, we have 100 such $\mathbf{k}$ filters in a convolutional layer, activation maps will be $\mathbf{o} \in \mathbb{R}^{220\times220\times100}$. After convolution, we applying an element-wise non-linearity, σ (typically ReLU), to the activation maps.

An activation map is computed from local responses of the same filter over the entire spatial locations. In other words, we use the same filter weights over the input tensor, which results in a large decrease in learnable parameters. This phenomenon is referred to as weight sharing in the context of CNN. Coming back to our example, the total learnable parameters in the convolutional layer with 100 filters would be approximately 7500 (= $5 \,.\, 5 \,.\, 3 \,.\, 100$). Which is much smaller (≈ 2000 times less) than using a fully connected layer with 100 neurons, which has approximately 1.50×10^7 learnable parameters. That said, it is within the capacity of a fully connected neural network layer to learn convolution by tying weights of all neurons in an appropriate structure. This, in turn, results in most parameters being zero which is a huge waste of computation, and as mentioned previously training such a network would be memory intensive, even computationally intractable.

Summary and implementation details. We have so far discussed the mechanism behind the convolutional layer. By stacking such convolutional layers in a cascade setting, we get a convolutional neural network. When implementing a convolutional layer, there are four hyperparameters that we need to specify. The height $\mathbf{k}_h$ and width $\mathbf{k}_w$ of the filter, number of filters m, stride s (amount to shift the filter), and the amount of padding P (typically add zeros) on the borders of the input tensor. Now let's show a general formulation of a convolutional layer:

I. **Input.** The input to the convolutional layer is a 3-D tensor $\in \mathbb{R}^{h_i\times w_i\times d_i}$

II. **Parameter Estimation.** Since we have m filters in the layer, the total learnable

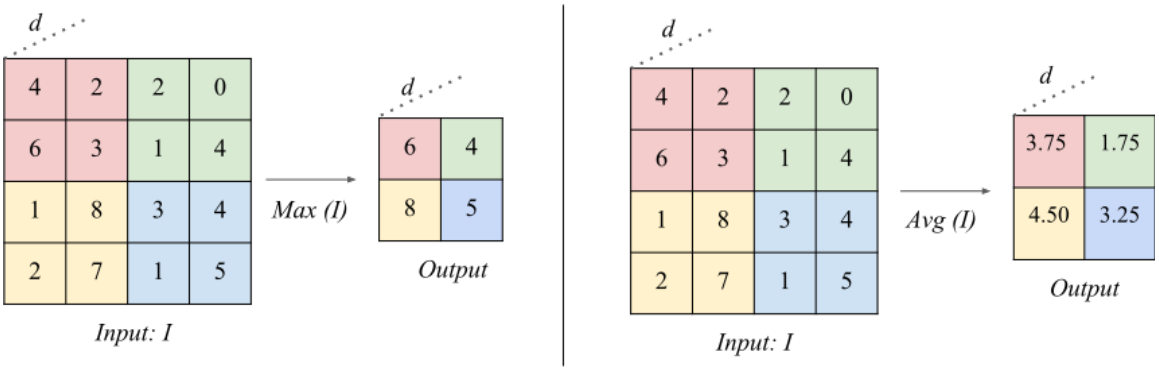

Figure 2.4: **Left.** Illustration of max-pooling with kernel $\mathbf{k} \in \mathbb{R}^{2\times2}$ ($P = 0$, $s = 2$). **Right.** Illustration of average-pooling with kernel $\mathbf{k} \in \mathbb{R}^{2\times2}$ ($P = 0$, $s = 2$). Both pooling mechanisms are performed independently for each activation map d_i in the input tensor. Figure courtesy [20].

parameters are: $m \,.\, (k_h \,.\, k_w \,.\, d_i) + m$ (include biases).

III. **Output.** The output of a convolutional layer is also a 3-D tensor of size: $h_o \times w_o \times m$, where [20]:

$$h_o = \frac{h_i - k_h + 2P}{s} + 1 \quad ; \quad w_o = \frac{w_i - k_w + 2P}{s} + 1 \tag{2.5}$$

Pooling layer. Pooling is used to downsample the activation maps. Often, a convolutional layer is followed by a pooling layer, but not always. To an extent applying pooling introduces translation-invariance into the network, meaning the pooling layer output does not change if the input values are shifted spatially by a small amount. Downsampling the activation maps further reduces the computational complexity of the network, as subsequent layers operate on smaller input tensor [19]. That said, it is possible to downsample in the convolutional layer itself by increasing the stride of the convolution. Currently, this method of downsampling is favored over pooling, especially when training good generative models, such as variational autoencoders (VAEs) or generative adversarial networks (GANs) [20].

Like convolution, we have a fixed window (filter) size that is slide across the in-

put tensor. Pooling is a 2-D operation, performed independently for each activation map (depth slice) in the input tensor. For example, let $\mathbf{I} \in \mathbb{R}^{220\times220\times100}$ be an input activation map to the pooling layer. Assuming our filter size is $\mathbf{k} \in \mathbb{R}^{2\times2}$ and is shifted spatially (stride) of 2. After pooling our output tensor will be $\mathbf{o} \in \mathbb{R}^{110\times110\times100}$, discarding around 50% of the activations. There are two kinds of pooling: *max-pooling* and *average-pooling*. In max-pooling, we take the maximum value from the input tensor filter block, whereas in average-pooling, the average of all values is computed. An illustration of both pooling mechanisms is shown in Figure 2.4.

Summary and implementation details. When implementing a pooling layer, there are two hyperparameters that we need to specify. The height p_h and width p_w of the filter, and stride s (amount to shift the filter). Note that one should be careful when deciding the filter size. Larger filter sizes are too destructive since we are discarding activations. The general formulation of a pooling layer:

I. **Input.** The input to the pooling layer is either a 3-D tensor $\in \mathbb{R}^{h_i\times w_i\times d_i}$ or 2-D tensor $\in \mathbb{R}^{h_i\times w_i}$.

II. **Parameter Estimation.** There are no parameters in the pooling layer.

III. **Output.** The output of a pooling layer is a downsampled 3-D tensor $\in \mathbb{R}^{h_o\times w_o\times d_i}$, where [20]:

$$h_o = \frac{h_i - p_h}{s} + 1 \quad ; \quad w_o = \frac{w_i - p_w}{s} + 1 \tag{2.6}$$

2.1.2 Recurrent Neural Networks

So far we have discussed neural networks that follow a feed-forward approach. There is no feedback that allows the network to make predictions based on previous inputs. Recurrent Neural Networks (RNNs) are a class of neural networks used to process sequential data. Input to a RNN is typically a sequence of length l containing vectors, expressed as $\{\mathbf{x}_1^\top, \mathbf{x}_2^\top, ..., \mathbf{x}_l^\top\}$. Unlike previous neural network

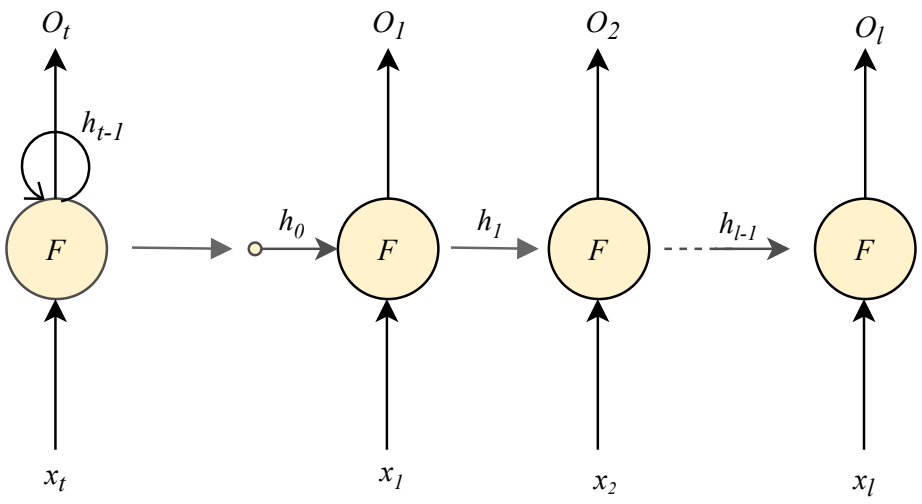

Figure 2.5: Block diagram of a RNN. It shows the RNN computation unfolded in time. Output at time t, O_t depends on the input $\mathbf{x}_t$ and previous state $\mathbf{h}_{t-1}$.

architectures, RNNs have a hidden state $\mathbf{h}_t$ that gets updated during each input of the sequence. The hidden state of the RNN can be interpreted as neural networks achieving a very primitive form of memory capability. The hidden state at each step is updated using the following recurrence formula:

$$\mathbf{h}_t = \mathcal{F}_{\mathbf{W}}(\mathbf{h}_{t-1}, \mathbf{x}_t) \tag{2.7}$$

where $\mathcal{F}$ is some RNN model and $\mathbf{W}$ are the shared weights and biases for every time step t. At time step t_1 the hidden state $\mathbf{h}_0$ is either initialized to 0 or treated as a learnable parameter [16]. An illustration of RNN is shown in Figure 2.5.

RNN implementation. Let the input to the RNN at each time step be $\mathbf{x}_t \in \mathbb{R}^d$, and the hidden state $\mathbf{h}_t \in \mathbb{R}^h$, then RNN is implemented as follows:

$$\mathbf{h}_t = \sigma(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{xh}\mathbf{x}_t + \mathbf{b}) \tag{2.8}$$

where, $\mathbf{W}_{hh} \in \mathbb{R}^{h\times h}$ and $\mathbf{W}_{xh} \in \mathbb{R}^{h\times d}$ are weight matrices used to transform previous hidden state $\mathbf{h}_{t-1}$ and input $\mathbf{x}_t$, $\mathbf{b} \in \mathbb{R}^h$ is the bias vector, and σ is some

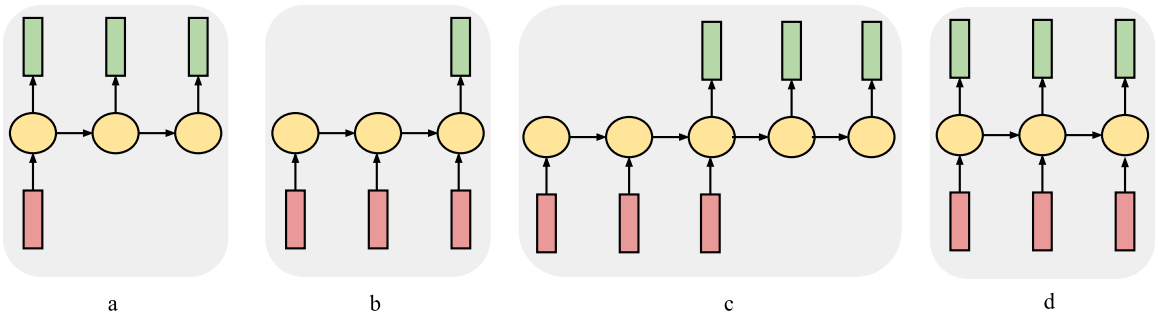

Figure 2.6: Block diagram of different RNN models. RNN's allows for sequential processing of vectors. This makes it useful for tasks that require sequential or time. **a.** Shows one input and many output model. **b.** Shows many input and one output model. **c.** This model maps many inputs to many outputs in a serial fashion. **d.** This model maps many inputs to many outputs in parallel fashion. Figure is based on [20].

non-linearity typically used are tanh or ReLU. Different RNN models are shown in Figure 2.6. The total learnable parameters in RNN are: $h \,.\, h + h \,.\, d + h$.

A major challenge with RNN is, it can be very challenging to learn long-term dependencies. Two key challenges are the *exploding-gradient* and *vanishing-gradient* problems. Exploding-gradients occurs when the gradients become too large. This can be often mitigated by clipping the gradients to be at a certain range. Unlike Exploding-gradients, Vanishing-gradients are challenging to rectify. Let's demostrate vanishing-gradient problem with a simplified recurrence relation of the RNN, given as:

$$\mathbf{h}_t = \mathbf{W}^\top \mathbf{h}_{t-1} \tag{2.9}$$

where $\mathbf{W}$ are the weights of the RNN. Note that there is no non-linear activation function or an input $\mathbf{x}_t$ [19]. Such repetitive multiplication of the weights is analogous to the power method algorithm used to find the largest eigenvalue and its corresponding eigenvector of a matrix [21]. Using this principle, Equation 2.9 can be re-written as: $\mathbf{h}_t = \mathbf{W}^\top h_0$. Assuming $\mathbf{W}$ has an eigendecomposition, $\mathbf{W} = \mathbf{Q}\Lambda\mathbf{Q}^{-1}$,

the reccurance formulation is given as:

$$\mathbf{h}_t = \mathbf{Q}^\top \Lambda^t \mathbf{Q}^{-1} \mathbf{h}_0 \tag{2.10}$$

Since the principal eigenvalues are raised to the power of t, eigenvalues whose magnitude is less than one will vanish [19]. In practice for RNN's we use non-linear activation functions such as $tanh$, which places all values between 1 or -1. The derivative of $tanh$ is zero at both ends (saturation region), so if the weights have small values, from Equation 2.10 we can see that the gradients will shrink exponentially.

Long-Short Term Memory Networks

Long-Short Term Memory Networks (LSTMs) is a variant of RNN, used in deep architectures specifically used to address the vanishing-gradient problem. Unlike RNN, rather than applying an element-wise nonlinearity to the affine transformation of inputs and recurrent units, LSTM consists of *gates* that have an internal recurrence (a self-loop), in addition to the outer recurrence of the RNN [19]. These gates enable LSTM's to both accumulate and forget states conditioned on the context. LSTM block diagram is shown in Figure 2.7.

LSTM mechanism. There are two states in LSTM, cell-state $\mathbf{c}_t$ and hidden-state $\mathbf{h}_t$. LSTM consists of three gates called the forget gate, the input gate, and the output gate. The forget gate decides what information is going to be kept in the cell-state and the input gate decides on which values to update in the cell-state. Outputs from both forget gate and input gate are used to update the cell-state. After which the output from the output gate and the updated cell-state are used to update the hidden-state $\mathbf{h}_t$.

LSTM implementation. Let inputs to the LSTM at each time step be $\mathbf{x}_t \in \mathbb{R}^d$, previous hidden state $\mathbf{h}_{t-1} \in \mathbb{R}^h$, and previous cell state $\mathbf{c}_{t-1} \in \mathbb{R}^h$, then LSTM is

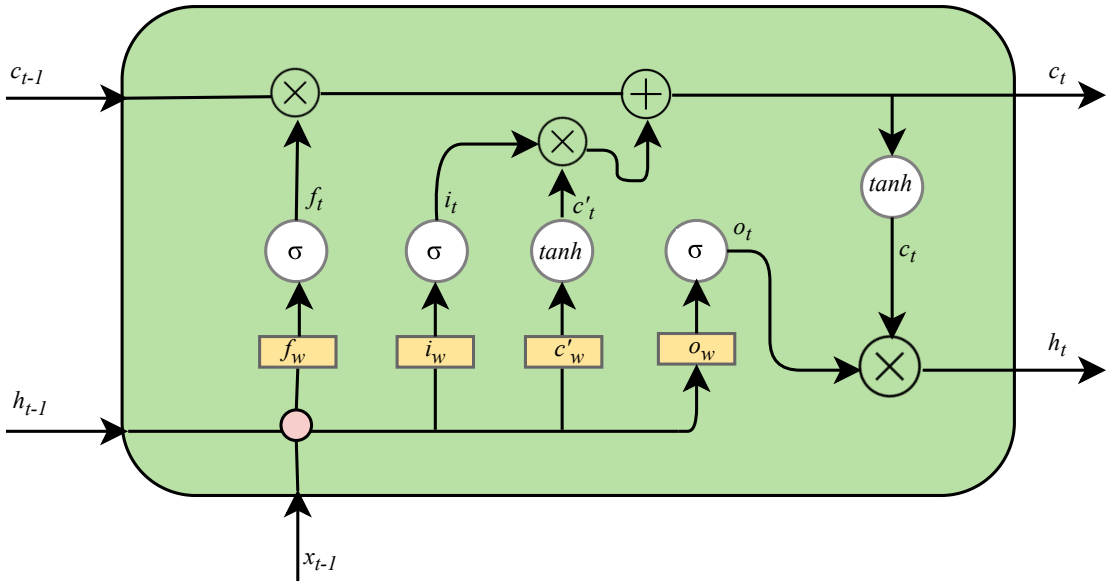

Figure 2.7: Block diagram of LSTM model. Figure is adapted from [22].

implemented as follows:

$$
\begin{aligned}
\mathbf{f}_t &= \sigma_g(\mathbf{W}_f\mathbf{x}_t + \mathbf{U}_f\mathbf{h}_{t-1} + \mathbf{b}_f) \\
\mathbf{i}_t &= \sigma_g(\mathbf{W}_i\mathbf{x}_t + \mathbf{U}_i\mathbf{h}_{t-1} + \mathbf{b}_i) \\
\mathbf{o}_t &= \sigma_g(\mathbf{W}_o\mathbf{x}_t + \mathbf{U}_o\mathbf{h}_{t-1} + \mathbf{b}_o) \\
\mathbf{c}_t &= \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \sigma_c(\mathbf{W}_c\mathbf{x}_t + \mathbf{U}_c\mathbf{h}_{t-1} + \mathbf{b}_c) \\
\mathbf{h}_t &= \mathbf{o}_t \circ \sigma_h(\mathbf{c}_t)
\end{aligned}
\tag{2.11}
$$

where operator $\circ$ denotes the Hadamard product, $\mathbf{f}_t$ is the forget gate, $\mathbf{i}_t$ is the input gate, $\mathbf{o}_t$ is the output gate, and $\mathbf{c}_t$ and $\mathbf{h}_t$ are the updated cell states. $\mathbf{W}_{(f,i,o,c)} \in \mathbb{R}^{h\times d}$ and $\mathbf{U}_{(f,i,o,c)} \in \mathbb{R}^{h\times h}$ are the associated weight matrices. $\mathbf{b}_{(f,i,o,c)} \in \mathbb{R}^h$ is the bias vector. The non-linearities used are: σ_g is sigmoid, σ_c and σ_h is typically tanh but can be replaced with ReLU. The total learnable parameters in LSTM is: $4\,.\,(h\,.\,d{+}h\,.\,h{+}h)$.

Convolutional LSTM. In this work we used a variant of LSTM called convolutional LSTM (ConvLSTM)) proposed by Shi *et al.* [23]. An advantage of using

ConvLSTM is that the network captures spatiotemporal correlations better, at the same time reduce the number of parameters. This is similar to using CNN over the fully-connected network. In ConvLSTM, the fully connected layers in the LSTM, are converted into fully convolutional layers.

Convolutional LSTM implementation. Let inputs to the ConvLSTM at each time step be $\mathbf{X}_t \in \mathbb{R}^{h_i \times w_i \times d_i}$, previous hidden state $\mathbf{H}_{t-1} \in \mathbb{R}^{h_h \times w_h \times d_h}$, and previous cell state $\mathbf{C}_{t-1} \in \mathbb{R}^{h_c \times w_c \times d_c}$, then ConvLSTM is implemented as follows:

$$\begin{aligned}
\mathbf{i}_t &= \sigma(\mathbf{W}_{xi} * \mathbf{X}_t + \mathbf{W}_{hi} * \mathbf{H}_{t-1} + \mathbf{W}_{ci} \circ \mathbf{C}_{t-1} + \mathbf{b}_i) \\
\mathbf{f}_t &= \sigma(\mathbf{W}_{xf} * \mathbf{X}_t + \mathbf{W}_{hf} * \mathbf{H}_{t-1} + \mathbf{W}_{cf} \circ \mathbf{C}_{t-1} + \mathbf{b}_f) \\
\mathbf{C}_t &= \mathbf{f}_t \circ \mathbf{C}_{t-1} + \mathbf{i}_t \circ \tanh(\mathbf{W}_{xc} * \mathbf{X}_t + \mathbf{W}_{hc} * \mathbf{H}_{t-1} + \mathbf{b}_c) \\
\mathbf{o}_t &= \sigma(\mathbf{W}_{xo} * \mathbf{X}_t + \mathbf{W}_{ho} * \mathbf{H}_{t-1} + \mathbf{W}_{co} \circ \mathbf{C}_t + \mathbf{b}_o) \\
\mathbf{H}_t &= \mathbf{o}_t \circ \tanh(\mathbf{C}_t)
\end{aligned} \tag{2.12}$$

where the operator $\circ$ denote the Hadamard product and $*$ denotes the convolution operator. $\mathbf{f}_t$ is the forget gate, $\mathbf{i}_t$ is the input gate, $\mathbf{o}_t$ is the output gate, and $\mathbf{C}_t$ and $\mathbf{H}_t$ are the updated cell states.

2.2 Training Neural Networks

Having established neural network models, the question still remains how do we get the weights $\mathbf{W}$ (neural network weights include bias too, for brevity we write both weights $\mathbf{w}$ and bias $\mathbf{b}$ together as $\mathbf{W}$). These weights are either learned from scratch or fine-tune (transfer learning) to perform the task at hand. In this work, we are concerned with supervised learning. In supervised learning, we have training dataset of samples which are assumed to be independent and identically distributed (i.i.d.) coming from a same distribution F. Each sample n has an input x_n,

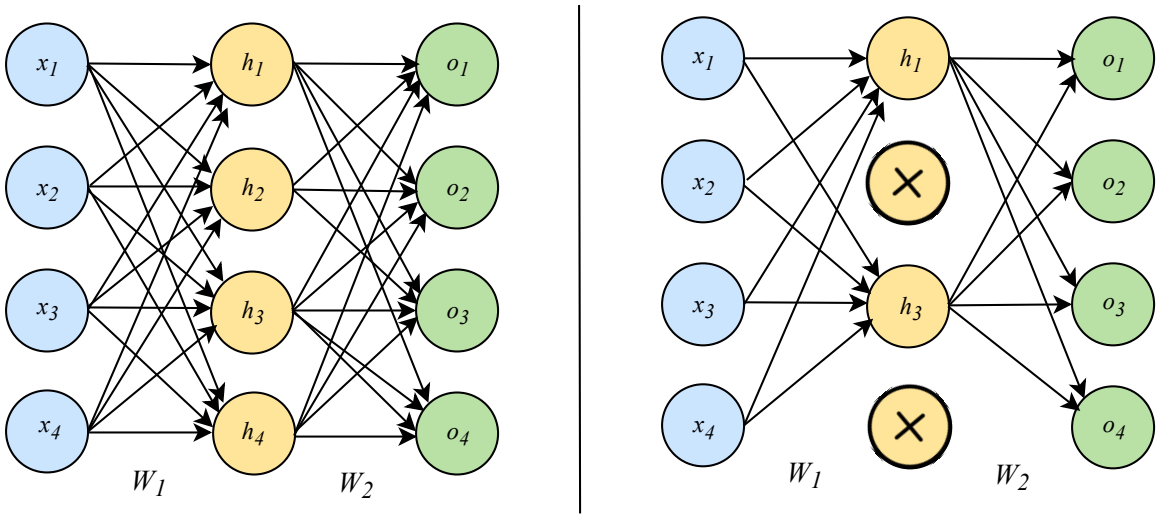

Figure 2.8: Illustration of dropout technique applied to a two-layer neural network. **Left.** Neural network without dropout applied. Each input is connected to every neuron in each layer. **Right.** Dropout applied to the hidden layer of the neural network. The connections to and from the dropped neurons are disabled. Figure is adapted from [24].

along with an associated ground truth, y_n. The primary objective in learning the weights of the neural network model is to minimize some loss L, between model prediction $\hat{y_n}$ and ground truth y_n as shown in Equation 2.13. Therefore, the final objective is shown in Equation 2.14, which states that we estimate $\mathbf{W}$ that minimizes the expected loss between $\hat{y_n}$ and y_n.

$$\mathcal{L} = L(\hat{y_n}, y_n) \tag{2.13}$$

$$J = \arg\min_{\mathbf{W}} \mathbb{E}[\mathcal{L}] \tag{2.14}$$

Equation 2.14 holds when we have access to all possible samples from the given distribution F. This is almost never the case, therefore finding an optimal f is intractable. Given that we only have finite samples N, under the i.i.d. assumption we can approximate the expected loss in Equation 2.14 as the average of the loss over

N such samples:

$$J \approx \arg\min_{\mathbf{W}} \frac{1}{N} \sum_{n=0}^{N} L(\hat{y_n}, y_n) \tag{2.15}$$

In practice, once we have found a function f that has optimal weights $\mathbf{W}$ of our network, we can discard the training data and only keep the learned weights. Depending on the task, the loss function in Equation 2.13 used, is described in Section 4.5.

Often neural network weights $\mathbf{W}$ can be much larger compared to the training samples N available. This can result in the network memorizing the training samples and performing poorly on previously unseen data. This problem is referred to as over-fitting. To address this problem, we add a regularization $\mathcal{R}$ to the weights of the network. The objective loss function is shown in Equation 2.16. We discuss two common techniques used in regularization of network weights: (1) L2 regularization and (2) Dropout.

$$J \approx \arg\min_{\mathbf{W}} \frac{1}{N} \sum_{n=0}^{N} L(\hat{y_n}, y_n) + \mathcal{R}(\mathbf{W}) \tag{2.16}$$

L2 regularization. This technique is used by neural networks and non-neural network machine learning models. It prevents the parameters from being too large by adding a penalty to the parameters. This helps in preventing some weights from having an excessive effect on predictions, thereby providing better generalization. The objective loss function with regularization function is:

$$J \approx \arg\min_{w,b} \frac{1}{N} \sum_{n=0}^{N} L(\hat{y_n}, y_n) + \lambda \|\mathbf{w}\|_2^2 \tag{2.17}$$

where λ is a hyperparameter specifying the strength of the regularization. In Equation 2.17, weights closer to zero will have little impact on loss, whereas large weights (outliers) can have a huge (*squared*) impact. In practice we tend to exclude the bias

terms from regularization, since they are only responsible for offsetting the model, and do not interact multiplicatively with inputs [20].

Dropout. This method was introduced by Srivastava *et al*. [24]. It reduces overfitting by randomly turning off neurons in a layer. When dropout is applied to a particular layer it independently turns off each neuron in that layer with a probability between 0 and 1. Such an approach prevents the network from co-adapting. The Figure 2.8 shows the visualization of dropout applied to a two-layer neural network. In practice, we apply this technique only to the fully connected layers of the neural network model and not to convolutional layers. The weights of the convolutional layer are highly correlated due to weight sharing, and randomly dropping them provides little to no effect on the overall model performance.

2.2.1 Optimization

In this section, we look at how to solve the optimization problem:

$$\arg\min_{\mathbf{W}} J(\mathbf{W}; \hat{y_n}, y_n) \tag{2.18}$$

in other words, we are minimizing some loss function with respect to the network weights. Before going more into optimization, it is worth-while to look at some prelimanaries. In supervised training, we typically learn the network weights from some dataset (depending on the task), where each input has an associated ground truth. A typical dataset is divided into training set, validation set, and testing set (in some cases there is no explicit validation set, typically a small percent taken from training set is used as the validation set). Note that when using SGD we require the gradient of a function. All the neural network models discussed in this work are end-to-end differentiable.

Coming back, in this section we will discuss the most common and so far an

effective technique used to train neural networks, the *Gradient Descent* (GD). This method belongs to the class of optimization algorithms known as *first order methods*. These methods use first order derivatives (or gradients), which shows the direction to move along in the search space.

In order to optimize using GD, we first need to obtain the gradients of the function with respect to it's weights: $\nabla_{\mathbf{W}} J(\mathbf{W}; \hat{y_n}, y_n)$. This gradient gives the direction to step in order to achieve a lower $L(\hat{y_n}, y_n)$. Therefore the network weights are updated as follows:

$$\mathbf{W} = \mathbf{W} - \alpha \nabla_{\mathbf{W}} J(\mathbf{W}; \hat{y_n}, y_n) \tag{2.19}$$

where α is the learning rate which controls the amount of step taken in the direction of the gradient. This procedure is repeated until convergence (when $L(\hat{y_n}, y_n)$ on validation/testing set is less than some error margin, ϵ).

The gradients in Equation 2.19 are computed using the entire training samples. In practice, training samples in a dataset can be very large i.e. Places dataset [25] has ≈ 9 million images and ImageNet dataset [26] has ≈ 1.2 million training images. Fitting all the data in a single step is not practical, hence we estimate the gradients using only a small subset of the samples. We split the training data into n subsets called batches. We obtain n by dividing total training samples by batch-size. This version of GD is called *Stochastic Gradient Descent* (SGD). Typically *batch size* is determined based on available memory, which means to do a single pass through the whole dataset will require n steps. Performing this single pass is called an *epoch*. It usually takes multiple such epochs for a model to converge.

Note that neural networks are highly non-convex functions, therefore before we begin training, it is important to initialize network weights ($\mathbf{W}$). As a first thought, it would be tempting to initialize all the weights to zero. This is a mistake as the network ends up computing the same output, the same gradients and undergo same weights update at every neuron. This is useless as we are not learning distinct

high-level features in every layer. Therefore, it is common to initialize the weights randomly, typically from a Gaussian with zero mean and small standard deviation (≤ 1.0) [20]. Such initializations help break the symmetry, which in turn allows the network to learn distinct features at different layers. There are also advanced initialization schemes such as *He* initialization [27], and *Xavier* initialization [28] which help networks converge faster.

Having discussed the optimization strategy, an important question arises, if we are able to reach an *optimal minimum*? (in other words, can we learn an optimal set of weights that will achieve the lowest $L(\hat{y_n}, y_n)$ on the testing set). There is no guarantee that an optimal minimum will be reached, and in some case even get stuck at a local minima such as saddle points. These saddle points are surrounded by high error plateaus that make it hard for the SGD to move out off, making learning slow down drastically [29]. Therefore, more advanced versions of SGD is employed in order to further improve the convergence speed, as discussed below.

The first version to SGD is the addition of *momentum*. Momentum helps accelerate SGD in the direction of the gradient, by accumulating an exponentially decaying moving average of past gradients. SGD with momentum is formulated as follows:

$$\begin{aligned} v_t &= \gamma v_{t-1} - \alpha \nabla_{\mathbf{W}} J(\mathbf{W}; \hat{y_n}, y_n) \\ \mathbf{W} &= \mathbf{W} + v_t \end{aligned} \tag{2.20}$$

where v_t is the velocity of the system, and $\gamma \in [0, 1)$ is the momentum term. The hyperparameter γ determines how rapidly the contributions from the previously accumulated gradients exponentially decay. In other words, it dampens velocity of the systems as it reaches a minimia, otherwise it would not stop. Note that compared to SGD, in SGD-momentum step size is the largest when gradients point in the same direction. There are also more incremental methods that have been intro-

duced recently, which adapts the learning rates of the network parameters. These include the AdaGrad [30], RMSProp [31], and Adam [32]. For more information on optimizers refer to [19]. In this work, we used Adam optimizer to train our neural network models.

In Adam, momentum is integrated directly into the first-order moments (the mean) of the gradient. Another key difference in Adam compared to other advanced first order optimizers is the inclusion of bias corrections to both the first-order and the second-order running moments. The final formulation of Adam is shown below:

$$\begin{aligned}
\widetilde{\mathbf{W}} &= \mathbf{W} + \beta_1 v_{t-1} \\
g &= \nabla_{\mathbf{W}} J(\widetilde{\mathbf{W}}; \hat{y_n}, y_n) \\
r_t &= \beta_2 r_{t-1} + (1-\beta_2) g \odot g \\
v_t &= \beta_1 v_{t-1} - \frac{\alpha}{\sqrt{r_t}} \odot g \\
\mathbf{W} &= \mathbf{W} + v_t
\end{aligned} \tag{2.21}$$

where, the hyperparameters, β_1 is the momentum coefficient, β_2 is the decay rate, and α is the learning rate. Adam is largely regarded as robust to the selection of hyperparameters (recommended to set $\beta_1 = 0.9$ and $\beta_2 = 0.999$ as their default values), with the exception of learning rate which needs to be adjusted [19].

2.2.2 Backpropagation

In order to use SGD we require the gradient $\nabla_{\mathbf{W}} J(\mathbf{W}; \hat{y_n}, y_n)$. Computing this gradient expression can be computationally intractable, since we are dealing with neural networks with a very large number of parameters. Alternatively, this gradient can be obtained numerically using the backpropagation algorithm, often referred

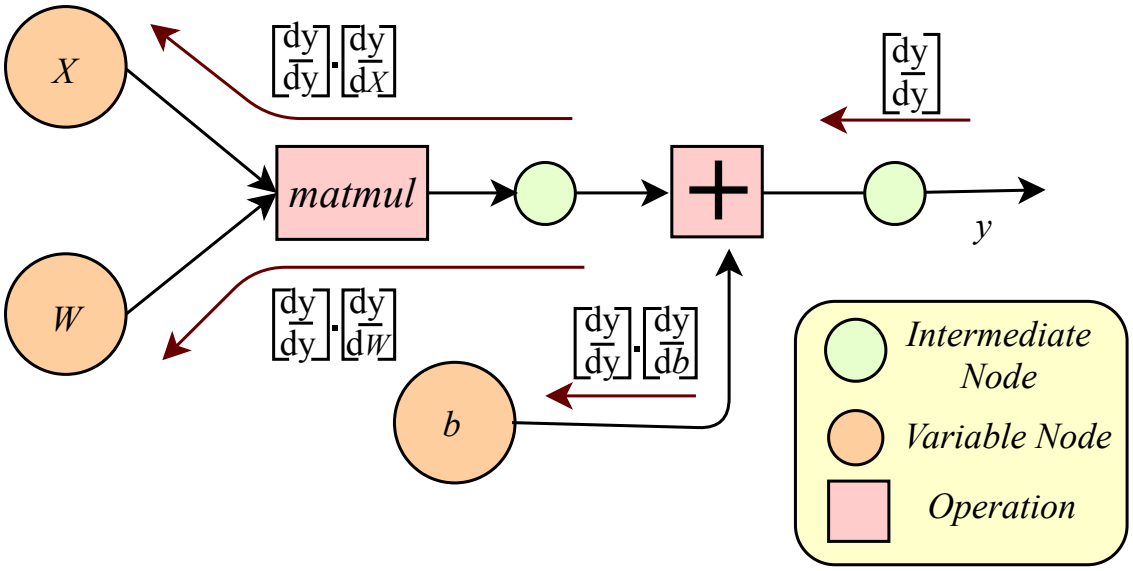

Figure 2.9: Artificial neuron expressed as computational graph and gradients computed using backprop.

to as *backprop* in deep learning. Backprop is a dynamic-programming algorithm which computes the gradient using simple and inexpensive procedure [19]. Before we dwell into the details of the backprop, let's introduce two more concepts: *chain rule* and *computational graph*.

Chain Rule. The chain rule of calculus is used to compute derivaties of functions by decomposing them into other functions whose derivatives are known [19]. Example, let $x \in \mathbb{R}$ and let f and g be two functions such that $f : \mathbb{R} \mapsto \mathbb{R}$ and $g : \mathbb{R} \mapsto \mathbb{R}$. Suppose $u = g(x)$ and $v = f(u)$, by chain rule:

$$\frac{\partial v}{\partial x} = \frac{\partial v}{\partial u}\frac{\partial u}{\partial x} \tag{2.22}$$

Note that chain rule can generalize to vectors as well. Example, let $x \in \mathbb{R}^m$, $y \in \mathbb{R}^n$ and let f and g be two functions such that $f : \mathbb{R}^m \mapsto \mathbb{R}^n$ and $g : \mathbb{R}^n \mapsto \mathbb{R}$. Suppose

$u = g(x)$ and $v = f(u)$, by chain rule:

$$\nabla_x v = (\nabla_u v)^\top \nabla_x u \tag{2.23}$$

where $\nabla_x u$ is $n \times m$ Jacobian matrix of g. Likewise for each operation in the graph the backprop algorithm computes the Jacobian-gradient product. Note that in practice we apply backprop to tensors of arbitary dimensions.

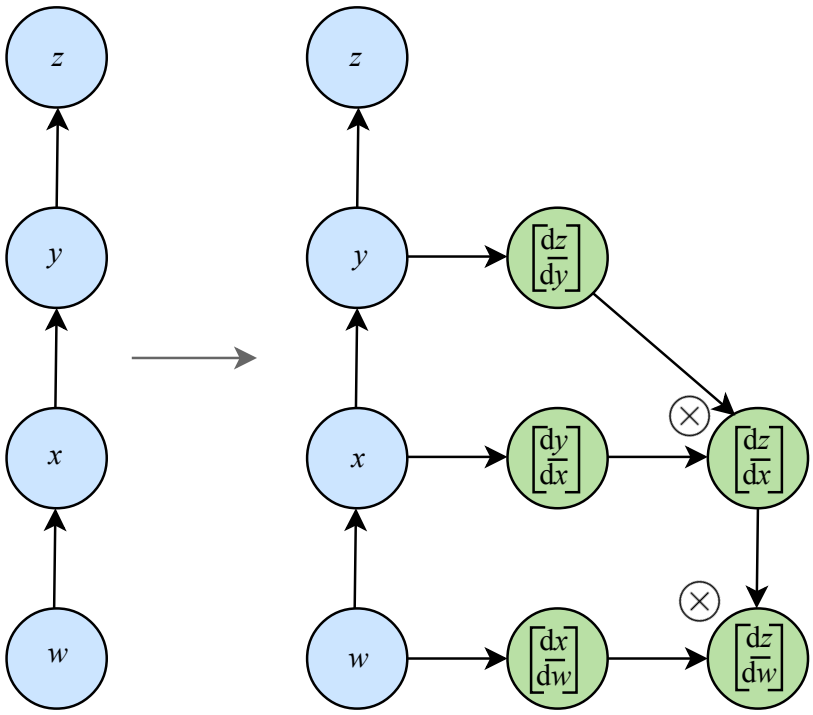

Figure 2.10: An example of symbol-to-symbol approach to computing derivatives using backprop. In this approach backprop adds notes to the computational graph on how to compute the derivatives. In this example, the graph is constructed by running backprop to compute $\frac{dz}{dw}$ expression. This figure is adapted directly from[19].

Computational Graph. Neural networks can be formally represented as a directed computational graph. Expressing neural networks as computational graphs help describe backpropagation algorithm more precisely [19]. Each node in the graph represents a variable which could be scalar, matrix, tensor, etc. Also, the graph consists of set of allowable *operations*. An operation will return the output

variable. An example of neuron expressed as a computational graph is shown in Figure 2.9. Modern deep learning frameworks, like *TensorFlow* [33], use symbol-to-symbol differentiation, where it creates additional nodes to the computational graph which provides a symbolic representation of required gradient operations. An advantage of casting the gradients itself as a computational graph makes it convenient to obtain higher order derivatives by backprop. An illustration of a symbolic approach is shown in Figure 2.10.

2.2.3 Training Procedure

Now that we have described neural networks and optimization techniques, let's apply these to formulate a training procedure that is typically used while training neural networks. A sample classification code in Tensorflow framework is shown in Listing 2.1.

2.3 CNN models

Deep convolutional network require large amounts of training data. This can be bottleneck in many tasks which have only limited training data. An example would be, in the ImageNet image classification task which has at least 1000 training images per class, whereas in MIT-67 indoor-scene classification which consists of 80 training images per scene. In practice, we typically train on a large dataset and fine-tune on the smaller dataset. This approach is called transfer learning.

There are a number of standard CNN models primarily trained on ImageNet such as AlexNet, VGG-$16/19$, GoogleNet, ResNet, etc. These can be used as an off-the-shelf feature extractor, or a starting point to fine-tune on the task dataset. In this work, we primarily used GoogleNet [3]. The key innovation in the GoogleNet architecture compared to previous incarnations such as AlexNet or VGG is the in-

troduction of inception module.

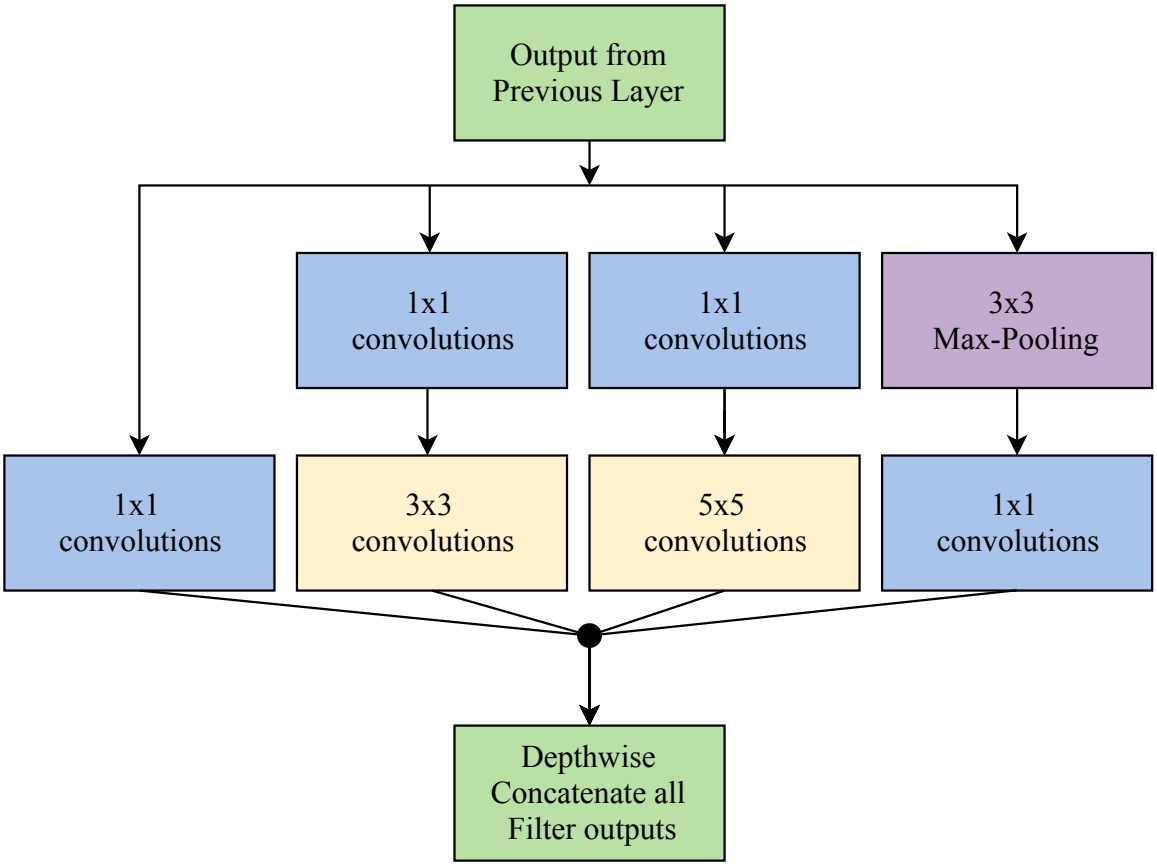

Figure 2.11: Block diagram of a Inception Module. Figure is adapted from [3]

The main motivation behind Inception architecture is to use filter-level sparsity at the same time maintaining translation invariance. Inception architecture achieves this by using convolutional layers as building blocks [3]. Inception module consists of multiple convolution layers operating on the same input tensor. Then the output of all these individual layers is concatenated along the depth dimension to form a single output tensor. The block diagram of the inception block is shown in Figure 2.11.

The GoogleNet architecture consists of five major convolutional blocks. The first two blocks are a single convolutional layer followed by the last three blocks which are inception modules. Counting individual layers in each block of GoogleNet to-

tals to 22 layers. All the outputs after convolutions, including those inside the Inception modules an element-wise non-linearity of ReLU is applied. Note that even the lower layers in the GoogleNet can be converted into inception modules. In practice, due to memory constraints, it was convenient to use Inception modules only at higher layers while the lower layers were kept in traditional convolutional form [3].

GoogleNet is a relatively deep network, therefore due to vanishing gradient problem discussed in Section 2.1.2 could affect the propagation of gradients. Therefore two auxiliary classifiers are connected to intermediate layers as shown in Figure 2.12 to combat the vanishing gradient problem while providing regularization [3]. During training, the auxiliary classifiers loss is added to the total loss of the network with a discount factor (the auxiliary classifiers losses were weighted by 0.3) [3]. At inference, the auxiliary networks are discarded and only the output from the final layer is used.

```python
#!/usr/bin/env python
# Import necessary packages
import tensorflow as tf
from googlenet import GoogleNet
from datasets import imagenet

class ImageNet_Classification(object):
    def __init__(self, data, input_data, label_data, **kwargs):
        self.data        = data
        self.images      = input_data
        self.labels      = label_data
        self.n_epochs    = kwargs.pop('n_epochs', 20)
        self.batch_size  = kwargs.pop('batch_size', 64)
        self.l_rate      = kwargs.pop('l_rate', 0.0001)

        # Setup model
        self.model

    def model(self):
        """
        This function defines the network along with the loss function
        """
        # GoogleNet Model
        net = GoogLeNet({'data': input_data})
        # Get output from last layer
        net_output = net.layers['FC_1000']
        # Loss
        self.loss = tf.nn.softmax_cross_entropy_with_logits(labels=
            label_data, logits=net_output)

    def train(self, model, l_rate, n_epochs):
         """
        This function traines the network using SGD.
        """
        ## Select an SGD optimizer
        optimizer = tf.train.AdamOptimizer(learning_rate=self.l_rate)

        ## Generate the gradient computational graph using backprop.
        grads = tf.gradients(self.loss, tf.trainable_variables())

        ## Generate the gradient computational graph with parameter
            update.
        grads_and_vars = list(zip(grads, tf.trainable_variables()))
        train_op = optimizer.apply_gradients(grads_and_vars=
            grads_and_vars)

        # Train Data Loader
        train_loader = self.data.gen_data_batch(self.batch_size)

        # Compute steps
        n_examples        = self.data.max_steps
        n_iters_per_epoch = int(np.ceil(float(n_examples)/self.
            batch_size))
```

```
50
51          # Start a tensorflow session
52          with tf.Session() as sess:
53              ## Intialize the training graph
54              sess.run(tf.global_variables_initializer())
55
56              ## Begin training
57              # for epoch in epochs
58              for e in range(self.n_epochs):
59                  # epoch loss variable
60                  alv = 0
61                  # for batch in batches
62                  for i in range(n_iters_per_epoch):
63                      # Sample n data pairs from training set
64                      image_batch, label_batch = next(train_loader)
65                      feed_dict = {self.images: image_batch,
66                                   self.labels: label_batch}
67                      _, interm_loss = sess.run([train_op, loss],
                            feed_dict)
68                      alv += interm_loss
69
70                  if (alv/n_examples) < epsilon:
71                      break
72  #-------------------------------
73  def main():
74      # Training parameters
75      epsilon = 0.01
76      epochs  = 1000
77      batch_size = 128
78      learning_rate = 0.0001
79
80      # Place-Holders to feed data into graph
81      images = tf.placeholder(tf.float32,[None, 224, 224, 3],name='imgs')
82      labels = tf.placeholder(tf.float32,[None, 1000],name='labels')
83
84      # Load Model
85      ImgClsfy = ImageNet_Classification(data=imagenet, input_data=images,
            label_data=labels, n_epochs=epochs, batch_size=batch_size,
           l_rate=learning_rate)
86
87      # Begin Training
88      ImgClsfy.train()
89  #-------------------------------
90  if __name__ == "__main__":
91      main()
```

Listing 2.1: Sample tensorflow classification code illustrating Algorithm 1 in python

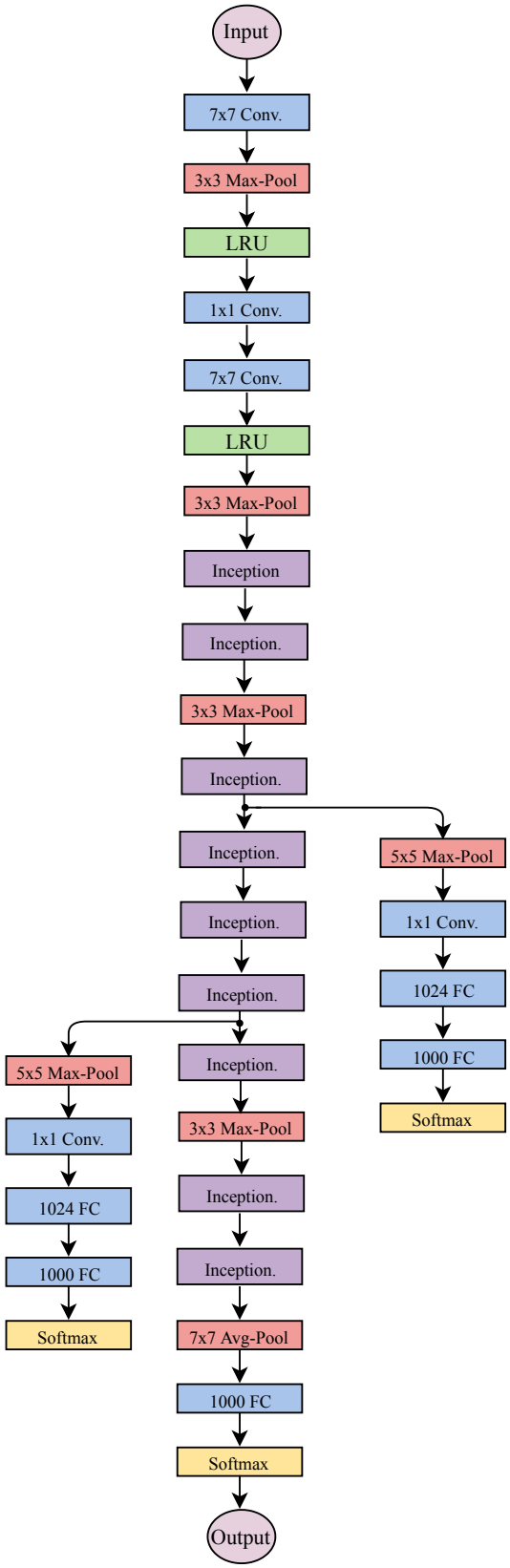

Figure 2.12: Block diagram of a GoogleNet architecture. The inception block referes to the inception module shown in Figure 2.11. Figure is adapted from [3]

CHAPTER 3

Related Works

In this chapter we will look at the related works on attention and two tasks that we apply attention too: Image-based camera pose regression and indoor scene classification. These two problems, we believe, provide ideal test cases to study the model of attention presented in this paper. Consider, for example, the problem of camera pose estimation. It is conceivable that the system first focuses on the overall scene, which is captured by higher levels of abstraction, and next attends to fine details captured by lower levels of abstraction, such as existence of a window.

3.1 Attention

Attention is a mechanism that dynamically allocates computational resources to the most salient features of the input signal. Attention has appeared in a variety of recent architectures [34–40]. A natural way to implement a sequential attentional probing mechanism is with a Recurrent Neural Network (RNN) or variant (e.g., Long Short-Term Memory, LSTM [35, 41]) in conjunction with a gating function

[23, 42, 43] that yields a soft (e.g., softmax or sigmoid) or hard attention [16, 44]. The attentional policy is learned without an explicit training signal, rather the task-related loss alone provides the training signal for the attention-related weights. In this work, we incorporate both soft (spatial selection) and hard (layer selection) attention in an end-to-end trainable architecture. Most closely related to the current work are the soft and hard selection mechanisms proposed by Xu *et al*. [16] and Veit and Belongie [40], respectively. Xu *et al*. [16] proposed an end-to-end trainable soft spatial attention architecture for image captioning. We adapt this soft attention architecture for our purposes and further extend it to include hard attention. Veit and Belongie [40] proposed a dynamic convolutional architecture that selects whether or not information propagates through a given CNN layer during a forward pass. Similar to Veit and Belongie [40], we use the recently proposed Gumbel-Softmax to realize our discrete (hard) selection of layers.

3.2 Image-based camera pose regression

Low-level features (e.g., SIFT [45]) have dominated the camera pose localization literature, e.g., [46–49]. An early example of using high-level features for camera localization appeared in Anati *et al*. [46], where heatmaps from object detections were used for localization. More recently, high-level CNN features have garnered attention. These features can be considered as soft proxies to object detections. Kendall *et al*. [50,51] proposed PoseNet, an image-based 6-DoF camera localization method. PoseNet regresses the camera position and orientation based on input provided by a CNN layer. Kendall and Cipolla [52] reconsidered the loss used in PoseNet to integrate additional geometric information. Walch *et al*. [2] extended the PoseNet approach by introducing an LSTM-based dimensionality reduction step prior to regression to avoid overfitting. In each case, the networks rely on features from a

manually selected layer, located relatively high in the feature hierarchy. In contrast, we propose an attentional network that is capable of dynamically integrating the most salient features across the spectrum of feature abstractions (capturing potentially texture-like and object-related features as necessary).

3.3 Indoor scene classification

To demonstrate the generality of our approach we also consider a classification task, indoor scene classification. Here, a wealth of research has considered both handcrafted (e.g., [53,54]) and learned deep features, e.g., [18,55]. In this work, we compare our approach using a standard deep architecture, GoogLeNet [3], which we also use as the base network for our layer-spatial attention method.

CHAPTER 4

Methodology

The proposed layer-spatial attention network sequentially probes the input signal over a fixed number of steps. It is comprised of a hard selection mechanism that selects a CNN layer (Sec. 4.3) and soft attention that selects a spatial location within the selected layer (Sec.4.1). The attention network is realized using a convolutional LSTM (Conv-LSTM)[23]. Figure 4.4 provides an overview of our architecture. At each Conv-LSTM step, the layer attention selects a CNN layer and spatial attention localizes a region within it. After N recurrent steps, the Conv-LSTM hidden states for all steps are concatenated and used for classification or regression.

4.1 Where: Spatial attention

We adapt the soft attention mechanism from Xu *et al*. [16] as the foundation of our method, with a key difference that is, we used convolutional layers instead of fully-connected. At each time step t, the input to the attention layer consists of deep feature (specifically, the feature map) from (the currently selected) CNN layer plus the

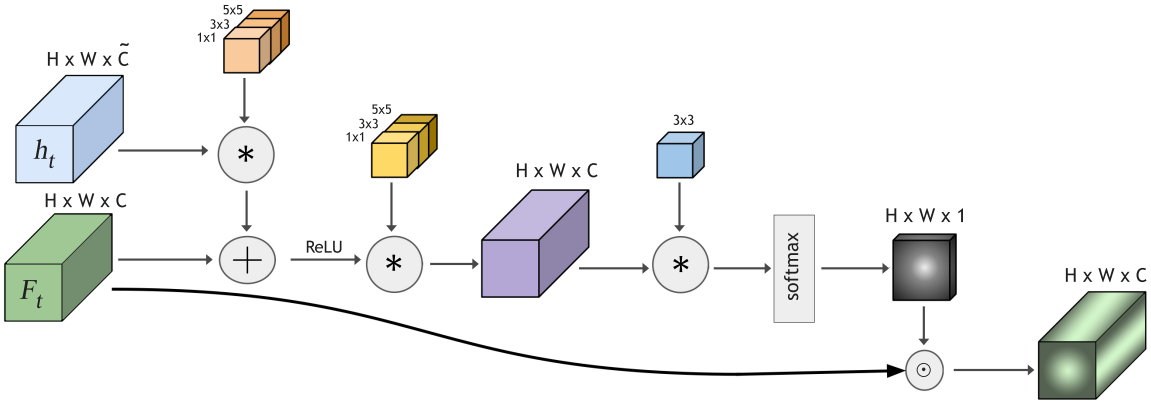

Figure 4.1: Multi-Convolutional Soft Attention Mechanism.

LSTM hidden state from the previous time step. At each time step, the attention mechanism selects a feature it deems most likely to improve task performance.

4.2 Soft Attention implementation.

At each time step t, the spatial attention mechanism receives as input the selected layer $\mathbf{f} \in \mathbb{R}^{h_f \times w_f \times d_f}$ (see Section 4.3) and the recurrent hidden state $\mathbf{h}_t \in \mathbb{R}^{h_h \times w_h \times d_h}$ from the previous step. The soft attention module is implemented as follows:

$$\begin{aligned} \mathbf{h}_{att} &= \mathbf{h}_t * \mathbf{C}_h \\ \mathbf{f}_{att} &= \mathbf{ReLU}(\mathbf{h}_{att} + \mathbf{f}) \\ \mathbf{A} &= \mathbf{f}_{att} * \mathbf{C}_A \\ \mathbf{A}_{\mathbf{mask}} &= \mathbf{softmax}(\mathbf{A}) \\ \mathbf{O}_{att} &= \mathbf{A}_{\mathbf{mask}} \odot \mathbf{f} \end{aligned} \tag{4.1}$$

where $*$ denotes the convolutional operator and $\odot$ is element-wise multiplication. $\mathbf{E}_h$ and $\mathbf{C}_\mathbf{A}$ are two convolutional layers, which compute an embedding and (un-

scaled) attention mask, respectively. The embedding layer $\mathbf{E}_h$ is used to transform the hidden state $\mathbf{h}_t$, channel dimension to equal to the input layer's channel dimension. The $\mathbf{C}_A$ layer computes the unscaled attention mask with dimensions $h_f \times w_f \times 1$. The final attention mask is computed by taking the $\mathrm{softmax}$ of the unscaled attention mask. The output of the attention layer $\mathbf{O}_{att}$ is obtained by taking an element-wise multiplication between the features in each channel and attention map.

During training, we also add an additional attention penality loss adapted from [16]. The penality loss is implemented as follows:

$$L_{mask} = \alpha \sum_{i=1}^{h_f w_f} (1 - \sum_{t=1}^{N} \mathbf{A}_{\mathbf{mask_t}}) \tag{4.2}$$

where α determines the weight of the regularizer. Since $\mathbf{A}_{\mathbf{mask}} = 1$, as it is an output from softmax. This loss penalizes the model if only one region in the image is only selected for N steps.

4.2.1 Multi-Convolutional Soft Attention mechanism.

Unlike the soft attention mechanism proposed in Xu *et al*. [16] our's replace fully-connected layers with convolutional layers. Specifically, we used multi-convolutional layers that uses different kernel sizes similar to an inception module.

At each time step t, the module receives h_t from ConvLSTM and the selected feature layer F_t. The ConvLSTMs hidden state h_t is first converted to the appropriate channel size of the feature map. We add the embedding h_t and feature layer F_t. Then we apply a non-linearity (Leaky ReLU). After which we compute the attention weights and apply softmax to get the attention map. Then an element-wise multiplication is performed between features and attention map to get the final output of the soft attention module. The Multi-ConvLSTM is applied to attention output. At

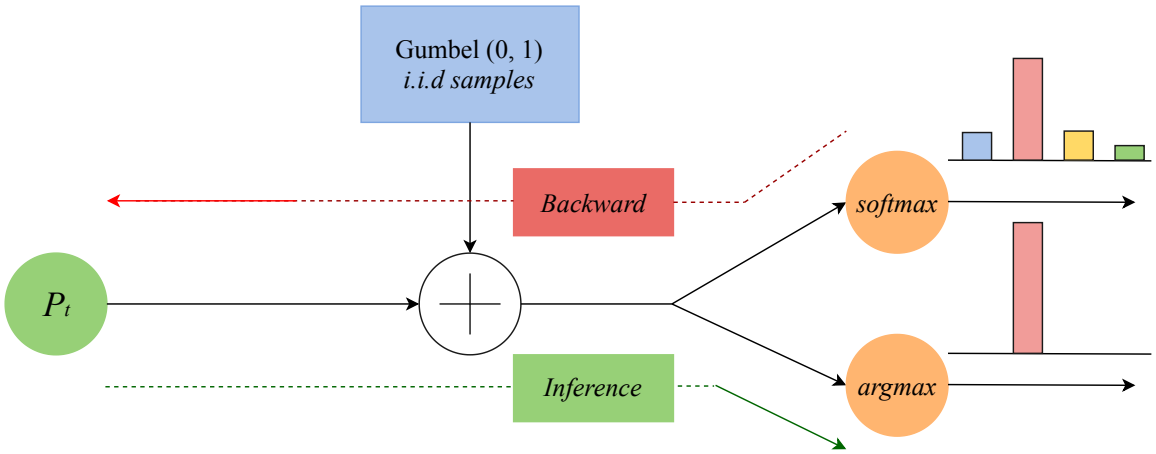

Figure 4.2: Gumbel-softmax represented as computation graph. Figure adapted from[40].

each time step the LSTM output is used for prediction. In Section 5.4 we experimentally show the perfomance again obtained using Multi-Convolutional approach. Figure 4.1, illustrates the multi-convolutional soft attention mechanism.

4.3 What: Layer attention

In layer attention (i.e., "what" features to attend) a CNN layer is selected whose feature map is deemed to contain the most salient information at the current recurrent step. Our layer attention involves a discrete (hard) selection of a CNN layer. Here, we use the recently proposed continuous relaxation of the Gumbel-Max trick [56], the Gumbel-Softmax [57, 58], to realize the discrete selection of layers.

Gumbel-Max provides a simple and efficient way to draw samples from a categorical (discrete) distribution:

$$z = \text{one\_hot}(\arg\max[g_i + \log \pi_i]), \tag{4.3}$$

where, $g_1, ..., g_k$ are i.i.d. samples drawn from the $\text{Gumbel}(0, 1)$ distribution, and π_i are unnormalized probabilities. Samples g are drawn using the following proce-

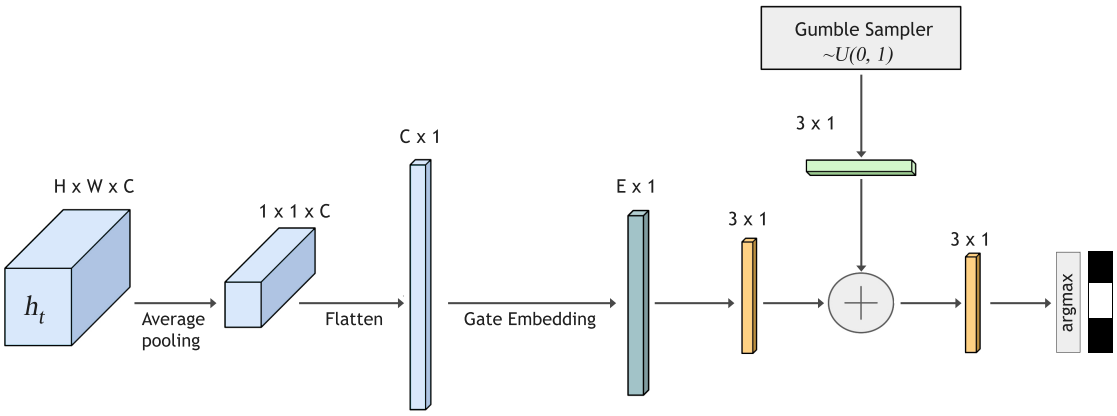

Figure 4.3: Layer Selection Mechanism.

dure: (i) draw sample $u \sim \text{Uniform}(0, 1)$; and (ii) set $g = -\log(-\log(u))$. In the forward pass (and during testing), we compute the arg max of the unnormalized log probabilities. In contrast, in the backward pass the $\arg\max$ is approximated with a softmax function:

$$y_i = \frac{\exp\left(\frac{\log(\pi_i)+g_i}{\tau}\right)}{\sum_{j=1}^{k}\exp\left(\frac{\log(\pi_j)+g_j}{\tau}\right)}, \tag{4.4}$$

where k is the number of CNN layers that are considered for selection, $i \in [1, k]$, and τ represents temperature. (This approach is the straight-through version of the Gumbel-Softmax estimator proposed in [58].) During training the temperature, τ, is progressively lowered. As the temperature approaches zero, samples from the Gumbel-Softmax distribution closely approximate those drawn from a categorical distribution. An illustration of gumbel-softmax method is shown in Figure 4.2.

For layer attention, we realize the (layer) selection scores (i.e., unnormalized probabilities) at each recurrent step as the output of a fully connected layer computed using the previous hidden state. During the forward pass we perform layer selection using Equation 4.3 and in the backward pass gradients are computed using Equation 4.4 to keep our architecture end-to-end trainable.

4.3.1 Layer selection mechanism architecture.

The mechanism receives input h_t from ConvLSTM. It then performs an average pool and an intermediate gate embedding before prediction. We add the Gumbel samples to the predicted logits and perform an $argmax$ to select the optimal layer. The gate embedding layer dimension E is much smaller than C. This gate embedding layer helps build a possible representation of incoming features at every LSTM steps, without significantly increasing the network parameters. Figure 4.3, illustrates the layer selection mechanism.

4.4 Joint Spatial and Layer Attention Architecture

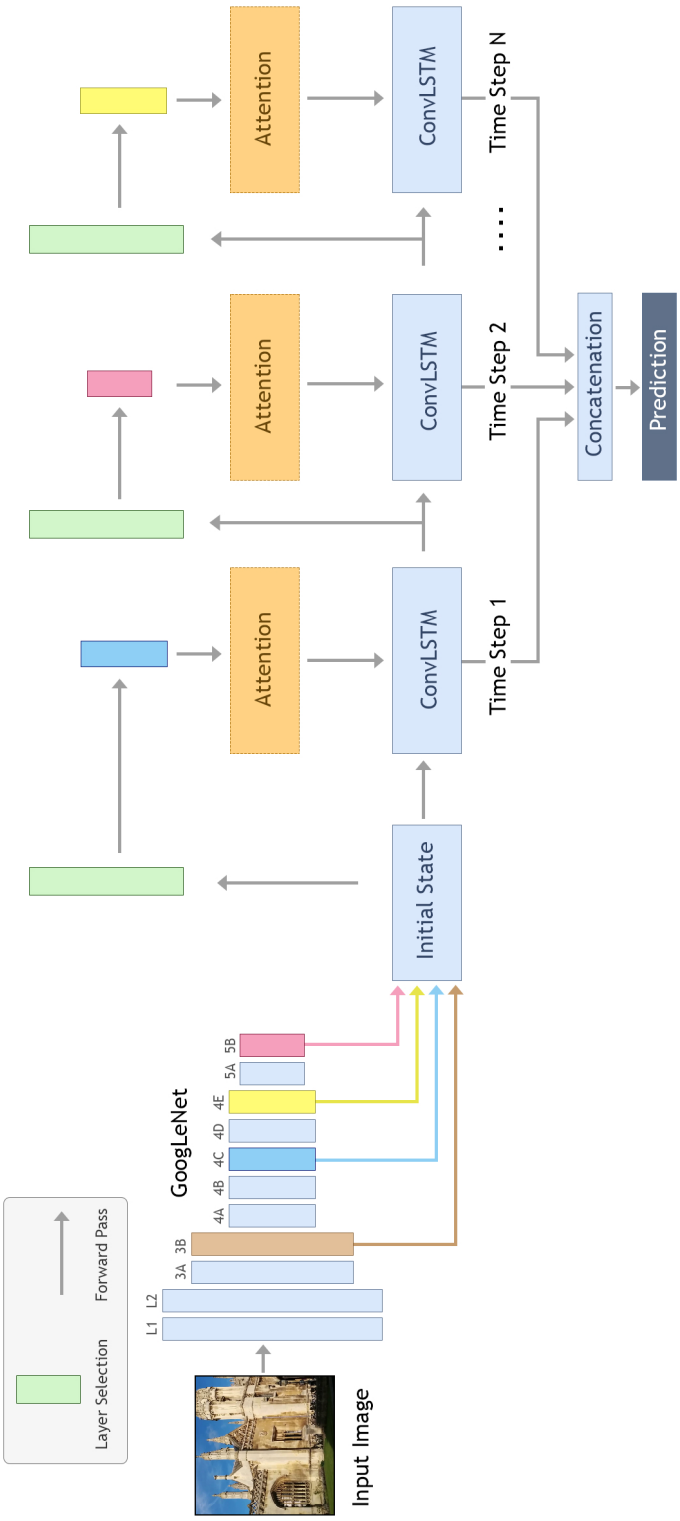

Figure 4.4: Overview of our layer-spatial attention architecture. Layer-spatial attention is realized within a Conv-LSTM framework, where the layer attention uses the previous hidden state, and spatial attention uses both the selected layer and the previous hidden state. After N Conv-LSTM steps, the hidden states from all steps are concatenated and used for regression or classification.

4.5 Tasks

In our approach, after N Conv-LSTM steps, the hidden states are concatenated, average pooled, and passed onto a fully connected layer for (regression/classification) prediction. To ensure that our comparisons are meaningful, and that any differences in the performance of our method to those posted by previous methods are due to our attention mechanism, we use the exact same losses as those used by our baselines.

4.5.1 Camera Pose Estimation

The proposed camera localization network takes an RGB image as input and outputs camera position and orientation $[\hat{\mathbf{x}},\ \hat{\mathbf{q}}]^\top$. Camera pose is defined relative to an arbitrary reference frame. We use the same regression loss as our baselines [2,50,59] to facilitate direct empirical comparison:

$$\mathcal{L} = \|\mathbf{x} - \hat{\mathbf{x}}\|_2 + \beta \|\mathbf{q} - \frac{\hat{\mathbf{q}}}{\|\hat{\mathbf{q}}\|_2}\|_2, \tag{4.5}$$

where $[\mathbf{x},\ \mathbf{q}]^\top$ represent ground truth position $\mathbf{x}$ and orientation $\mathbf{q}$, and $[\hat{\mathbf{x}},\ \hat{\mathbf{q}}]^\top$ denote predicted position $\hat{\mathbf{x}}$ and orientation $\hat{\mathbf{q}}$. Orientations are represented using quaternions. β is a scalar hyperparameter that determines the relative weighting between the positional and orientation errors. We use the same β value as our baselines, PoseNet [50] and LSTM-PoseNet [2].

4.5.2 Indoor Scene Classification

Consistent with our scene classification baseline [3], we use the standard cross-entropy classification loss:

$$\mathcal{L} = -\mathbf{y}_c^\top \log(\hat{\mathbf{y}}_c), \tag{4.6}$$

where $\mathbf{y}_c$ is a one-hot encoded class label for class c, and $\hat{\mathbf{y}}_c$ is the output of the softmax classifier.

CHAPTER 5

RESULTS

In this chapter we report and discuss the experimental results. In Section 5.1 we detail the experimental setup which includes the hyper-parameters. Section 5.2 we discuss the datasets used in this work. Section 5.3 shows results on the layer-spatial attention applied to camera pose estimation and indoor scene classification tasks. Finally, in Section 5.5 shows an ablation study using layer-spatial attention.

5.1 Experimental Setup

To realize our layer-spatial attention model we use the same basic architecture as Xu *et al*. [16] for sequential spatial attention. We augment this network with hard attention for layer selection. To avoid overfitting, we replace the LSTM layers with ConvLSTM [23] layers that reduce the network weight parameterization.

We used *TensorFlow* framework [33] to implement and train our models. Every model was trained end-to-end using ADAM [32] optimizer with the parameters: $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 1 \times 10^{-8}$. We used a learning rate equal to 1×10^{-4}. The

regularization parameter, $\lambda = 2 \times 10^{-4}$, was added to weights, but not to biases. The dropout probability was set to 0.5 for all the experiments. The LSTM hidden size was set to 96 for all experiments as well.

Camera Pose Estimation. Images were resized to 256×455 pixels. During training, we performed random crops of 224×224 pixels. At test time, we performed center crop of 224×224 pixels. The batch size was set to 40. Similar to [50] and [2], separate mean images were computed for each channel and the images were mean subtracted per channel. From the loss function in Equation 4.5, a balance β has to be used between the orientation and translation because they are regressed from the same model weights. Experimentally it was found that β is greater for outdoor scenes as position errors tended to be relatively greater. For Cambridge Landmarks dataset β value was set between 250 to 2000. For 7-Scenes dataset β value was set between 120 to 750, and for TUM-LSI dataset β value was set to 1000.

Indoor scene classification. We resized the images to 256×256. During training, we performed random crops of 224×224 pixels. At test time, we performed centered crop of 224×224 pixels. The batch size was set to 40. The images were mean subtracted per channel.

For both camera pose estimation and indoor scene classification, we used the same pre-trained CNN layers as used by previous methods. Specifically, we used the original GoogLeNet weights trained on Places[1] [25]. By necessity, we converted these provided trained network weights to be able to use these in *TensorFlow*.

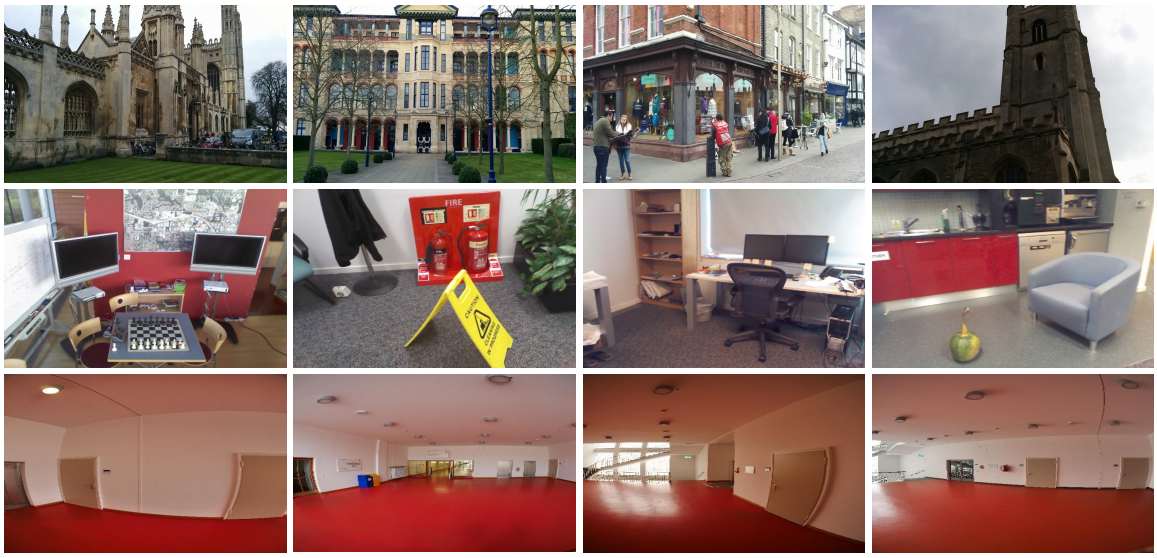

Figure 5.1: (a) Top row: Cambridge Landmarks Dataset. King's College, Old Hospital, Shop Facade and St. Mary's Church. (b) Middle row: 7-Scenes (subset). Chess, Fire, Office and Pumpkin. (c) Bottom row: TUM-LSI.

5.2 Datasets

5.2.1 Cambridge Landmarks

Cambridge Landmarks [50] is a large scale outdoor dataset, containing five outdoor datasets. For our experiments, we only use the four datasets that were used by [50] and [2]. The dataset consists of RGB images. Six degrees-of-freedom camera poses are provided for each image. The dataset was collected using a smart phone, and structure from motion was employed to label each image with its corresponding camera pose.

5.2.2 7-Scenes

7-Scenes [60] is a small scale indoor dataset, which consists of seven different scenes. These scenes were obtained using Kinect RGB-D camera, and KinectFusion[61] was

[1] http://places.csail.mit.edu/downloadCNN.html

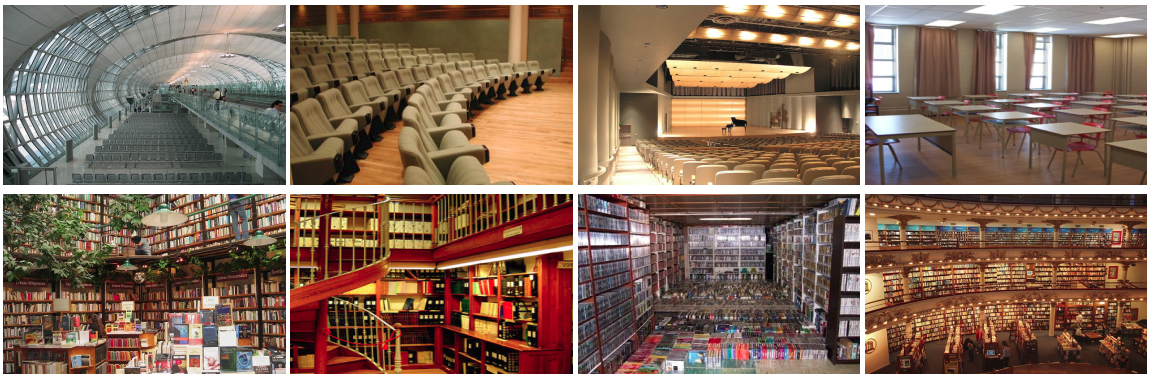

Figure 5.2: MIT-67 Indoor Scene Dataset. (a) Top row: Airport, Auditorium, Concert Hall and Classroom. A network can have a hard time classifying them by just focusing on specific properties, since all of them contain large hallways with chairs. (b) Bottom row: Bookstore, Library, Video Store and Library. This set of images have almost the same structure and objects which makes these scenes very ambiguous.

used to obtain the ground truth. We use the train/test split used by [50] and [2]. Scene contain ambiguous regions, which makes camera localization difficult.

5.2.3 TU Munich Large-Scale Indoor (TUM-LSI)

TUM-LSI [2] is an indoor dataset, which covers an area of two orders of magnitude larger than that covered by the 7Scenes dataset. It consists of 875 training images and 220 testing images. We use the train/test split used by [2]. This is a challenging dataset to localize due to repeated structural elements with nearly identical appearance.

5.2.4 MIT-67

MIT-67 [17] is an indoor scene dataset. Images taken primarily in four different indoor environments — store, home, public spaces, leisure and working places. The dataset contains 67 categories in total. We used the official train/test split provided by [17]. Each category has 80 training images and 20 testing images.

5.3 Experimental Results and Discussion

Figure 5.3 shows the frequencies of the GoogLeNet feature layers selected for each dataset on the respective test sets. As can be seen, the datasets predominately utilize more than one layer. Furthermore, the layers most frequently selected differ widely amongst the datasets.

We found that for image-based camera localization using three Conv-LSTM steps worked best, after which the performance decreases, the error increases. In the case of indoor scene classification two Conv-LSTM steps performed best..

5.3.1 Camera localization

Table 5.1 compares our proposed method against image-based camera pose regression methods [2, 50, 51]. All the compared methods use GoogLeNet as the source of features for regression, with the baselines limiting features to layer Conv-5B.

In terms of the individual scenes, our method achieves the least error in both translation and rotation in the majority of cases at three steps. Considering the aggregate results over the respective datasets, we see our method yields significant improvements over the previous methods, ranging between 12.3 and 25.1 percent for translation and 1.79 and 13.9 percent for rotation.

The TUM-LSI dataset contains large textureless surfaces and repetitive scene elements covering over $5,575\ m^2$. Active search or SIFT-based approaches have been previously shown to perform poorly on this dataset [2]. Our method achieves state-of-the-art performance, suggesting that the ability to attend to different CNN layers over successive LSTM steps helps.

Figure 5.4 (top row) shows qualitative results for camera localization. For outdoor scenes, it appears our attention mechanism captures both low-level (e.g., corners) and high-level structures (e.g., rooftops and windows).

Table 5.1: Camera localization results. Median localization error achieved by the proposed attention model over three steps on Cambridge Landmarks, 7-Scenes, and TUM-LSI. Bold values indicate the lowest error achieved for each row. Improvement is reported with respect to LSTM-PoseNet [2]. A dash (-) indicates that no result is reported.

Dataset	Area or Volume	PoseNet [50]	Bayesian PoseNet [59]	LSTM PoseNet [2]	Ours			
					Conv-LSTM Step-1	Conv-LSTM Step-2	Conv-LSTM Step-3	Improvement (meter, degree)
Great Court	8000 m^2	-	-	-	-	-	-	-
Kings College	5600 m^2	1.66 m, 4.86°	1.74 m, 4.06°	0.99 m, **3.65**°	1.02 m, 4.22°	1.00 m, 4.51°	**0.90** m, 3.70°	+9.09, -1.36
Old Hospital	2000 m^2	2.62 m, 4.90°	2.57 m, 5.14°	1.51 m, 4.29°	1.62 m, 4.11°	1.51 m, 4.02°	**1.36** m, **3.95**°	+9.93, +7.92
Shop Facade	875 m^2	1.41 m, 7.18°	1.25 m, 7.54°	1.18 m, 7.44°	1.15 m, 5.45°	0.95 m, 6.44°	**0.91** m, **5.29**°	+22.8, +28.8
St. Marys Church	4800 m^2	2.45 m, 7.96°	2.11 m, 8.38°	1.52 m, 6.68°	1.62 m, 7.22°	1.59 m, 5.94°	**1.42** m, **6.07**°	+6.57, +1.64
Street	50000 m^2	-	-	-	18.7m, 34.1°	15.0 m, 30.3°	**13.9** m, **30.0**°	-
Average [2]	3319 m^2	2.08 m, 6.83°	1.92 m, 6.28°	1.30 m, 5.52°	1.35 m, 5.25°	1.26 m, 5.22°	**1.14** m, **4.75**°	+12.3, +13.9
Chess	6.0 m^3	0.32 m, 6.08°	0.37 m, 7.24°	0.24 m, 5.77°	0.17 m, 5.58°	0.16 m, 5.27°	**0.15** m, **4.79**°	+37.5, +16.9
Fire	2.5 m^3	0.47 m, 14.0°	0.43 m, 13.7°	0.34 m, 11.9°	0.32 m, 12.6°	0.31 m, 11.7°	**0.23** m, **10.0**°	+32.3, +15.9
Heads	1.0 m^3	0.30 m, 12.2°	0.31 m, 12.0°	0.21 m, 13.7°	0.18 m, 13.8°	0.18 m, 14.1°	**0.18** m, **13.7**°	+14.2, +0.00
Office	7.5 m^3	0.48 m, 7.24°	0.48 m, 8.04°	0.30 m, 8.08°	0.29 m, 7.63°	0.29 m, 7.23°	**0.29** m, **8.02**°	+3.33, +0.74
Pumpkin	5.0 m^3	0.49 m, 8.12°	0.61 m, 7.08°	0.33 m, 7.00°	0.25 m, 5.46°	0.25 m, 5.76°	**0.26** m, **6.16**°	+21.2, +12.0
Red Kitchen	18 m^3	0.58 m, 8.31°	0.58 m, 7.51°	**0.37** m, 8.83°	0.43 m, 8.03°	0.37 m, 7.49°	0.39 m, **8.20**°	-2.00, +5.77
Stairs	7.5 m^3	0.48 m, 13.1°	0.48 m, 13.1°	0.40 m, 13.7°	0.32 m, 9.98°	0.31 m, 10.5°	**0.29** m, **12.0**°	+27.5, +12.4
Average All	6.9 m^3	0.44 m, 9.01°	0.46 m, 9.81°	0.31 m, 9.85°	0.28 m, 9.01°	0.26 m, 8.86°	**0.25** m, **8.98**°	+19.1, +9.10
TUM-LSI	5575 m^2	1.87 m, 6.14°	-	1.31 m, 2.79°	1.32 m, 3.82°	1.26 m, 3.69°	**0.98** m, **2.74**°	+25.1, +1.79

CNNaug-SVM [55]	S^2ICA [18]	GoogLeNet [3]	Ours			
			Conv-LSTM Step-1	Conv-LSTM Step-2	Conv-LSTM Step-3	Improvement (%)
69.0 %	71.2 %	73.7 %	74.5 %	**77.1** %	76.0 %	+3.4

Table 5.2: Mean accuracy results for indoor scene classification on MIT-67. The proposed method achieves the highest accuracy (shown in boldface). Improvement is reported with respect to the GoogLeNet [3] baseline.

5.3.2 Indoor scene classification

Table 5.2 compares our proposed layer-spatial attention method against three baselines [3,18,55]. The proposed method achieves best performance after two recurrent steps.

Figure 5.4 (bottom row) shows several qualitative results for indoor scene classification. The layer-spatial attention seems to capture objects and physical scene structures present in the scene. For the Concert Hall image, the attention mechanism appears to focus on the entire image, perhaps focusing on the scene architecture. For the Dental Office image, spatial attention picks out the dental equipment (a permanent fixture) and correctly ignores the person (a transient entity). For the Closet image, clothes and cabinetry are selected. Finally, for the Gym image, the proposed attention mechanism selects the exercise equipment.

5.3.3 Results for more Conv-LSTM steps

Camera localization. We did an experimental study for a subset of scenes from camera localization dataset shown in Table 5.3. We concluded that for the camera position estimation Conv-LSTM step three on average provides the best result.

Indoor Scene Classification. We did an experimental study on MIT-67 indoor scene, shown in Table 5.4. We concluded that for the Indoor Scene Conv-LSTM step two

Dataset	Area or Volume	PoseNet [50]	Bayesian PoseNet [59]	LSTM PoseNet [2]	Ours					
					Conv-LSTM Step-1	Conv-LSTM Step-2	Conv-LSTM Step-3	Conv-LSTM Step-4	Conv-LSTM Step-5	Improvement (meter, degree)
Old Hospital	2000 m^2	2.62 m, 4.90°	2.57 m, 5.14°	1.51 m, 4.29°	1.62 m, 4.11°	1.51 m, 4.02°	**1.36** m, **3.95**°	1.55 m, 4.46°	1.64 m, 4.20°	+9.93, +7.92
St. Marys Church	4800 m^2	2.45 m, 7.96°	2.11 m, 8.38°	1.52 m, 6.68°	1.62 m, 7.22°	1.59 m, 5.94°	**1.42** m, **6.07**°	1.49 m, 5.87°	1.58 m, 6.51 °	+6.57, +1.64
Office	7.5 m^3	0.48 m, 7.24°	0.48 m, 8.04°	0.30 m, 8.08°	0.29 m, 7.63°	0.29 m, 7.23°	**0.29** m, **8.02**°	0.29 m, 8.07°	0.30 m, 8.12 °	+3.33, +0.74
Stairs	7.5 m^3	0.48 m, 13.1°	0.48 m, 13.1°	0.40 m, 13.7°	0.32 m, 9.98°	0.31 m, 10.5°	**0.29** m, **12.0**°	0.31 m, 12.0°	0.33 m, 10.9 °	+27.5, +12.4
TUM-LSI	5575 m^2	1.87 m, 6.14°	-	1.31 m, 2.79°	1.32 m, 3.82°	1.26 m, 3.69°	**0.98** m, **2.74**°	1.14 m, 3.33°	1.18 m, 3.68 °	+25.1, +1.79

Table 5.3: Median localization error achieved by our proposed attention model over five-time steps on subset of Cambridge Landmarks, subset of 7-Scenes, and TUM-LSI. Bold values indicate the lowest error achieved for each row. Improvement is reported with respect to LSTM-PoseNet [2].

CNNaug-SVM [55]	S^2ICA [18]	GoogLeNet [3]	Ours					
			Conv-LSTM Step-1	Conv-LSTM Step-2	Conv-LSTM Step-3	Conv-LSTM Step-4	Conv-LSTM Step-5	Improvement (%)
69.0 %	71.2 %	73.7 %	74.5 %	**77.1** %	76.0 %	75.4	74.8	+3.4

Table 5.4: Mean accuracy results for indoor scene classification on MIT-67. The proposed method achieves the highest accuracy (shown in boldface). Improvement is reported with respect to the GoogLeNet [3] baseline.

on average provides the best result.

5.4 Multi-Convolutional Approach

In this section, we describe our motivation for using the multi-convolutional approach. To showcase how we arrived at the proposed approach, we provide evaluation on all three datasets for the pose estimation. We initially started with the same implementation as Xu *et al*. [16] for soft attention, by using fully connected layers. The model ended up overfitting the data and showed poor performance on the test set. Also, the network converged to select only a single spatial feature instead of probing through the other spatial features at different LSTM time-steps. Our first solution was converting fully connected layers into fully convolutional layers. The results for this approach on pose estimation is shown in Table 5.5. The results shown is quite far from [2] especially on the position, but interestingly median error was

Dataset	PoseNet [50]	LSTM-PoseNet [2]	Ours	
			Convolutional Spatial Attention	Improvement (meter, degree) %
King's College	1.66 m, 4.86°	**0.99 m**, 3.65°	1.39 m, **2.63**°	-27.2, +27.6
Old Hospital	2.62 m, 4.90°	**1.51 m**, 4.29°	3.72 m, **4.24**°	-120.5, +6.9
Office	0.48 m, 7.24°	**0.30 m**, 8.08°	0.64 m, **7.89**°	-103.3,+3.2
Stairs	0.48 m, 13.1°	**0.40 m**, 13.7°	0.48 m, **12.8**°	-15.0, +6.5
TUM-LSI	1.87 m, 6.14°	**1.31 m**, 2.79°	3.93 m, **2.15**°	+16, +22.9

Table 5.5: Median localization error achieved by the convolutional attention model on a subset of camera pose estimation datasets: Cambridge Landmarks, 7-Scenes, and TUM-LSI dataset. Bold values indicate the lowest error achieved for each row.

close to [50].

Dataset	PoseNet [50]	LSTM-PoseNet [2]	Ours	
			Multi-Conv. Spatial Attention	Improvement (meter, degree) %
King's College	1.66 m, 4.86°	0.99 m, **3.65**°	**0.95** m, 4.11°	+4.04, -12.6
Old Hospital	2.31 m, 5.38°	**1.51** m, **4.29**°	1.76 m, 4.44°	-16.5, -3.49
Office	0.48 m, 7.24°	0.30 m, 8.08°	**0.28** m, **7.52**°	+6.67, +6.93
Stairs	0.48 m, 13.1°	0.40 m, 13.7°	**0.32** m, **12.7**°	+20.0, +9.40
TUM-LSI	1.87 m, 6.14°	1.31 m, **2.79**°	**1.12** m, 3.66°	+14.5, -2.88

Table 5.6: Median localization error achieved by the multi-convolutional attention model on a subset of camera pose estimation datasets: Cambridge Landmarks, 7-Scenes, and TUM-LSI dataset. Bold values indicate the lowest error achieved for each row.

We found that our model was underfitting the training data. Naively increasing the depth size or kernel size was not showing any significant improvements. Therefore by taking inspiration from the inception module proposed in GoogLeNet [3], we converted each convolutional layer into multi-convolutional layers. We used three convolutional kernels with kernel sizes of 1x1, 3x3 & 5x5 and stacked their fi-

nal output together as shown in Figure 4.1. Similarly, in the case of ConvLSTM, we used four convolutional kernels with kernel sizes of 1x1, 3x3, 5x5 & 7x7. Then stacked their final output together for prediction. This approach helped improve results significantly as shown in Table 5.6. After which we applied our contribution of layer selection mechanism to form layer-spatial attention.

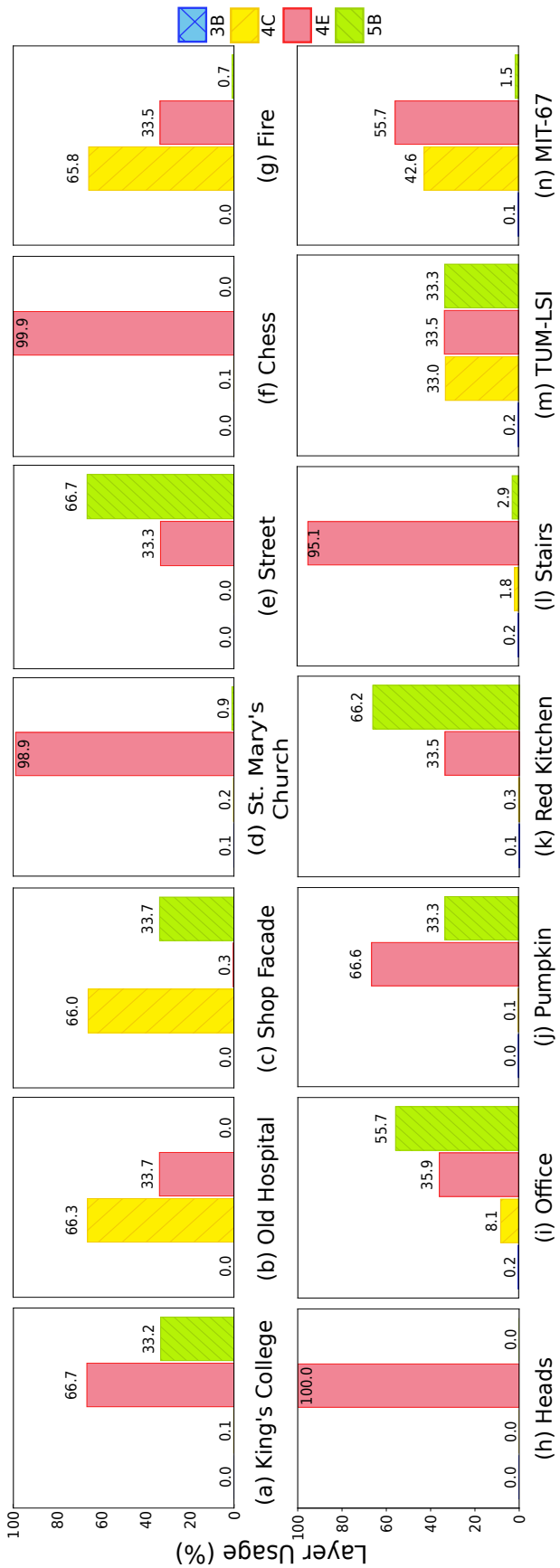

Figure 5.3: Layer Selection Frequencies (LSF) on all four datasets on the test set. (a) - (e) are Cambridge Landmarks scenes, (f) - (l) are scenes from 7-Scenes, (m) and (n) are TUM-LSI, and MIT-67 dataset, respectively. The bins refer to the GoogLeNet[3] Conv-{3B, 4C, 4E, 5B} layers. The vertical axis represents layer usage percentages.

Figure 5.4: Qualitative results on camera pose localization (top row) and indoor scene classification (bottom row). Top row: input image along with the spatial attention superimposed on the input image for three Conv-LSTM steps. Bottom row: spatial attention superimposed on the input image for two Conv-LSTM steps. The labels underneath each image indicates the selected CNN layer.

Dataset	Spatial Attention Only			Layer Selection Only	Spatial and Layer Attention
	Conv-3B	Conv-4E	Conv-5B		
Camera-Pose Estimation					
Old Hospital	1.49 m, 4.29°	1.42 m, 4.37°	1.76 m, 4.44°	2.36 m, 6.28°	**1.36** m, **3.95**°
Office	0.27 m, 7.37°	**0.26** m, **7.35**°	0.28 m, 7.52°	0.33 m, 7.97°	0.29 m, 8.02°
TUM-LSI	1.21 m, 3.26°	1.13 m, 3.66°	1.12 m, 3.66°	5.27 m, 10.8°	**0.98** m, **2.74**°
Indoor-Scene Classification					
MIT-67	61.6 %	74.5 %	74.2 %	76.4 %	**77.1** %

Table 5.7: Ablation study on layer-spatial attention. In all cases, GoogLeNet [3] Conv-{3B, 4E, 5B} layers are used. Bold values indicate the best result achieved for each row.

5.5 Ablation Study

Table 5.7 summarizes an ablation study that we performed to gauge the impact of combining layer selection with spatial attention. We choose Old Hospital (Cambridge Landmarks), Office (7-Scenes), TUM-LSI, and MIT-67 datasets for this ablation study. Old Hospital and Office were selected since we found these to be the most challenging for our proposed network.

We manually selected GoogLeNet's Conv-{3B, 4E, 5B} layers and applied spatial attention to each independently. (Note, the PoseNet results reported in Table 5.1 use layer Conv-5B without any form of attention for direct position-orientation regression.) Our results confirm that it is sometimes beneficial to use layers other then the final CNN layer. Median localization errors, for example, improve for both Old Hospital and Office datasets when we use layers other than Conv-5B. Note that in previous camera pose localization works [2, 50, 51] Conv-5B was manually selected. For indoor scene classification, selecting Conv-4E yields the best result. The last column of Table 5.7 includes results obtained by combining layer selection and spatial attention. Notice that in three out of four cases shown, network achieves best performance (lowest errors in case of camera pose estimation, and highest ac-

curacy in case of indoor scene classification) when using both layer selection and spatial attention. The second last column in Table 5.7 includes results when using layer selection alone. The network performance deteriorates when spatial attention is absent.

Our results are consistent with our initial guiding intuition that salient information is distributed across the spectrum of feature abstractions, e.g., things vs. stuff. Our proposed layer-spatial attention mechanism exploits this aspect to achieve better performance.

CHAPTER 6

Conclusion

This work presents a study of computational attention used in Convolutional Neural Networks (CNNs). The proposed model dynamically probes a set of convolutional layers of a CNN to process and aggregate the optimal set of features for a given task. Previously, a particular CNN layer is designate as the deep feature to be used during subsequent processing. Also, the entire features are typically processed to make a prediction. Our attention architecture learns to sequentially attend to different CNN layers (i.e., "what" feature abstraction to attend to) and different spatial locations of the selected feature map (i.e., "where") to perform the task at hand. This attention model learns completely from data without any additional supervisory signal. In Chapter 2 we provided some background on the deep learning used in this work and Chapter **4** covered our methodology.

In the context of computer vision, we demonstrated our approach on two computer vision tasks: (1) camera localization and (2) scene classification. We empirically showed that our approach of joint layer-spatial attention improves performance over manually selecting layers and previous approaches on both tasks.

6.1 Future Work

This work uses the same baseline feature extractor i.e. CNN as other previous work in order to provide a fair comparison. This way any improvement is coming from our approach and not better features. It is conceivable to think that using different feature extractor could potentially further improve the performance. In this work the layer selection is a hard selection, selecting only one layer at each reccurent time step. Another approach could be that using a combination of layers with weighting. The proposed approach to attention is general and may prove useful for other vision tasks i.e. single-image action recognition. As future work, it would be interesting to investigate with more complex layer (gating) selection network which can either be task dependent or could be more general.

Bibliography

[1] T. Joseph, K. Derpanis, and F. Qureshi. (2019) Joint spatial and layer attention for convolutional networks. [Online]. Available: https://arxiv.org/abs/1901.05376

[2] F. Walch, C. Hazirbas, L. Leal-Taixé, T. Sattler, S. Hilsenbeck, and D. Cremers, "Image-Based Localization Using LSTMs for Structured Feature Correlation," in *Proc. of the IEEE Conference on International Conference on Computer Vision (ICCV)*, 2017, pp. 627–637.

[3] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proc. of the IEEE Conference on Conference on Computer Vision and Pattern Recognition (CVPProc. MA,*, 2015, pp. 1–9.

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[5] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask r-cnn," in *Computer Vision (ICCV), 2017 IEEE International Conference on*. IEEE, 2017, pp. 2980–2988.

[6] L. Wang, T. Liu, G. Wang, K. L. Chan, and Q. Yang, "Video tracking using learned hierarchical features," vol. 24, no. 4. IEEE, 2015, pp. 1424–1435.

[7] E. Ilg, N. Mayer, T. Saikia, M. Keuper, A. Dosovitskiy, and T. Brox, "Flownet 2.0: Evolution of optical flow estimation with deep networks," in *IEEE conference on computer vision and pattern recognition (CVPR)*, vol. 2, 2017, p. 6.

[8] C. Ledig, L. Theis, F. Huszár, J. Caballero, A. Cunningham, A. Acosta, A. P. Aitken, A. Tejani, J. Totz, Z. Wang *et al.*, "Photo-realistic single image super-resolution using a generative adversarial network." in *CVPR*, vol. 2, no. 3, 2017, p. 4.

[9] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel, "Handwritten digit recognition with a back-propagation network," in *Advances in neural information processing systems*, 1990, pp. 396–404.

[10] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[11] C. Olah, A. Mordvintsev, and L. Schubert, "Feature Visualization," *Distill*, 2017, https://distill.pub/2017/feature-visualization.

[12] M. D. Zeiler and R. Fergus, "Visualizing and Understanding Convolutional Networks," in *Proc. of the European Conference on Computer Vision (ECCV)*, 2014, pp. 818–833.

[13] R. A. Rensink, "The Dynamic Representation of Scenes," *Visual Cognition*, vol. 7, pp. 17–42, 2000.

[14] J. K. Tsotsos, *A Computational Perspective on Visual Attention*. MIT Press, 2011.

[15] D. Bahdanau, K. Cho, and Y. Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate," in *Proc. of the International Conference on Learning Representations (ICLR)*, 2015.

[16] K. Xu, J. Ba, R. Kiros, K. Cho, A. C. Courville, R. Salakhutdinov, R. S. Zemel, and Y. Bengio, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention," in *Proc. of the International Conference on Machine Learning, (ICML)*, 2015, pp. 2048–2057.

[17] A. Quattoni and A. Torralba, "Recognizing indoor scenes," in *Proc. of the IEEE Conference on Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009, pp. 413–420.

[18] M. Hayat, S. H. Khan, M. Bennamoun, and S. An, "A Spatial Layout and Scale Invariant Feature Representation for Indoor Scene Classification," *Proc. of the IEEE Transactions on Image Processing*, pp. 4829–4841, 2016.

[19] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*. MIT press Cambridge, 2016, vol. 1.

[20] F.-F. Li, J. Justin, and S. Yeung. (2018) Course notes on cs231n: Convolutional neural networks for visual recognition. [Online]. Available: http://cs231n.stanford.edu/

[21] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," in *International Conference on Machine Learning*, 2013, pp. 1310–1318.

[22] C. Olah. (2015) Understanding lstm networks. [Online]. Available: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

[23] S. Xingjian, Z. Chen, H. Wang, D.-Y. Yeung, W.-K. Wong, and W.-c. Woo, "Convolutional lstm network: A machine learning approach for precipitation nowcasting," in *Advances in neural information processing systems*, 2015, pp. 802–810.

[24] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," vol. 15, no. 1. JMLR. org, 2014, pp. 1929–1958.

[25] B. Zhou, A. Lapedriza, J. Xiao, A. Torralba, and A. Oliva, "Learning deep features for scene recognition using places database," in *Proc. of Advances in Neural Information Processing Systems (NeurIPS)*, 2014, pp. 487–495.

[26] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[27] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.

[28] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 2010, pp. 249–256.

[29] Y. N. Dauphin, R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli, and Y. Bengio, "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization," in *Advances in neural information processing systems*, 2014, pp. 2933–2941.

[30] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," vol. 12, no. Jul, 2011, pp. 2121–2159.

[31] T. Tieleman and G. Hinton, "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude," *COURSERA: Neural networks for machine learning*, vol. 4, no. 2, pp. 26–31, 2012.

[32] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," in *Proc. of the International Conference on Learning Representations (ICLR)*, 2015.

[33] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A System For Large-Scale Machine Learning," *Proc. of the Operating Systems: Design and Implementation (OSDI)*, vol. 16, pp. 265–283, 2016.

[34] H. Larochelle and G. E. Hinton, "Learning to combine foveal glimpses with a third-order boltzmann machine," in *Advances in neural information processing systems*, 2010, pp. 1243–1251.

[35] W. Zaremba, I. Sutskever, and O. Vinyals, "Recurrent neural network regularization," 2014.

[36] Y. Tang, N. Srivastava, and R. R. Salakhutdinov, "Learning generative models with visual attention," in *Advances in Neural Information Processing Systems*, 2014, pp. 1808–1816.

[37] J. Ba, V. Mnih, and K. Kavukcuoglu, "Multiple Object Recognition with Visual Attention," in *Proc. of the IEEE Conference on International Conference on Robotics and Automation (ICRA)*, 2015.

[38] V. Mnih, N. Heess, A. Graves, and K. Kavukcuoglu, "Recurrent Models of Visual Attention," in *Proc. of Advances in Neural Information Processing Systems (NeurIPS)*, 2014, pp. 2204–2212.

[39] D. Jayaraman and K. Grauman, "Learning to look around: Intelligently exploring unseen environments for unknown tasks," *CoRR*, vol. abs/1709.00507, pp. 1–11, 2017.

[40] A. Veit and S. Belongie, "Convolutional networks with adaptive inference graphs," in *European Conference on Computer Vision*. Springer, 2018, pp. 3–18.

[41] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[42] M. F. Stollenga, J. Masci, F. J. Gomez, and J. Schmidhuber, "Deep Networks with Internal Selective Attention through Feedback Connections," in *Proc. of Advances in Neural Information Processing Systems (NeurIPS)*, 2014, pp. 3545–3553.

[43] F. Wang, M. Jiang, C. Qian, S. Yang, C. Li, H. Zhang, X. Wang, and X. Tang, "Residual Attention Network for Image Classification," in *Proc. of the IEEE Conference on Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 6450–6458.

[44] R. J. Williams, "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning," *Machine Learning*, vol. 8, pp. 229–256, 1992.

[45] D. G. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints," *International Journal of Computer Vision (IJCV)*, vol. 60, no. 2, pp. 91–110, 2004.

[46] R. Anati, D. Scaramuzza, K. G. Derpanis, and K. Daniilidis, "Robot Localization Using Soft Object Detection," in *Proc. of the IEEE Conference on International Conference on Robotics and Automation (ICRA)*, 2012, pp. 4992–4999.

[47] T. Sattler, B. Leibe, and L. Kobbelt, "Efficient & Effective Prioritized Matching for Large-Scale Image-Based Localization," *Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, vol. 39, no. 9, pp. 1744–1756, 2017.

[48] Y. Li, N. Snavely, D. Huttenlocher, and P. Fua, "Worldwide Pose Estimation using 3D Point Clouds," in *Proc. of the European Conference on Computer Vision (ECCV)*, 2012, pp. 15–29.

[49] J. Engel, T. Schöps, and D. Cremers, "LSD-SLAM: Large-scale direct monocular SLAM," in *Proc. of the European Conference on Computer Vision (ECCV)*, 2014, pp. 834–849.

[50] A. Kendall, M. Grimes, and R. Cipolla, "PoseNet: A Convolutional Network for Real-Time 6-DOF Camera Relocalization," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 2938–2946.

[51] A. Kendall and R. Cipolla, "Modelling uncertainty in deep learning for camera relocalization," in *Proc. of the IEEE Conference on International Conference on Robotics and Automation (ICRA)*, 2016, pp. 4762–4769.

[52] A. Kendall, R. Cipolla *et al.*, "Geometric loss functions for camera pose regression with deep learning," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 8–16.

[53] C. Doersch, A. Gupta, and A. A. Efros, "Mid-level visual element discovery as discriminative mode seeking," in *Advances in neural information processing systems*, 2013, pp. 494–502.

[54] M. Juneja, A. Vedaldi, C. Jawahar, and A. Zisserman, "Blocks that shout: Distinctive parts for scene classification," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2013, pp. 923–930.

[55] A. Sharif Razavian, H. Azizpour, J. Sullivan, and S. Carlsson, "Cnn features off-the-shelf: an astounding baseline for recognition," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition workshops (CVPR)*, 2014, pp. 806–813.

[56] E. J. Gumbel, *Statistical theory of extreme values and some practical applications: a series of lectures*. US Govt. Print. Office, 21954.

[57] C. J. Maddison, D. Tarlow, and T. Minka, "A* sampling," in *Proc. of Advances in Neural Information Processing Systems (NeurIPS)*, 2014, pp. 3086–3094.

[58] E. Jang, S. Gu, and B. Poole, "Categorical Reparameterization with Gumbel-Softmax," in *Proc. of the International Conference on Learning Representations (ICLR)*, 2017.

[59] A. Kendall and Y. Gal, "What uncertainties do we need in bayesian deep learning for computer vision?" in *Proc. of Advances in Neural Information Processing Systems (NeurIPS)*, 2017, pp. 5574–5584.

[60] J. Shotton, B. Glocker, C. Zach, S. Izadi, A. Criminisi, and A. Fitzgibbon, "Scene coordinate regression forests for camera relocalization in rgb-d images," in *Proc. of the IEEE Conference on Conference on Computer Vision and Pattern Recognition (CVPR)*, 2013, pp. 2930–2937.

[61] S. Izadi, D. Kim, O. Hilliges, D. Molyneaux, R. Newcombe, P. Kohli, J. Shotton, S. Hodges, D. Freeman, A. Davison *et al.*, "Kinectfusion: real-time 3d reconstruction and interaction using a moving depth camera," in *Proceedings of the*

24th annual ACM symposium on User interface software and technology. ACM, 2011, pp. 559–568.