11th January 2017

Masterarbeit im Fach Informatik
Rheinische Friedrich-Wilhelms-Universität Bonn
Institut für Informatik III
Computer Vision Group
Prof. Dr. J. Gall

# Convolutional Networks for Action Recognition

11th January 2017

vorgelegt von:
Mian Ahsan Iqbal
Matrikelnummer 2741933

Gutachter:
Prof. Dr. J. Gall
Prof. Dr. C. Bauckhage

Betreuer:
M. Sc. A. Richard

# Erklärung

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe. Alle Textauszüge und Grafiken, die sinngemäß oder wörtlich aus veröffentlichten Schriften entnommen wurden, sind durch Referenzen gekennzeichnet.

Bonn, 11th January 2017

Mian Ahsan Iqbal

# Abstract

Action recognition is a fundamental problem in computer vision with alot of potential applications such as video surveillance, human computer interaction and robot learning. Given pre-segmented videos, the task is to recognize actions happening within videos. Historically, hand crafted video features were used to address the task of action recognition. With the success of Deep ConvNets as image analysis method, a lot of extensions of standard ConvNets were purposed to process variable length video data. In this work, we propose novel recurrent ConvNet architecture called Spatio Temporal Residual networks to address the task of action recognition. The approach extends state of the art image analysis model called ResNet [He & Zhang$^+$ 15]. We show that on large-scale dataset, our model improves over the standard ResNet [He & Zhang$^+$ 15] architecture.

# Contents

# 1 Introduction

Action recognition in videos is of increasing research interest [Simonyan & Zisserman 14a,Feichtenhofer & Pinz$^+$ 16,Bilen & Fernando$^+$ 16] with many potential applications such as video surveillance, human computer interaction and robot learning. Action recognition is "the task of determining the actions happening in videos given pre-segmented videos". With the availability of large action recognition datasets such as UCF-101 [Soomro & Zamir$^+$ 12], it is usually addressed as a supervised learning task. Action recognition forms a basis for more sophisticated tasks such as classification of actions in untrimmed videos [Idrees & Zamir$^+$ 16], temporal detection of actions in videos [Richard & Gall 16, Yeung & Russakovsky$^+$ 15, Rohrbach & Amin$^+$ 12] and spatial localization of actions [Weinzaepfel & Martin$^+$ 16].

Traditionally, action recognition is addressed by hand crafted video features in combination with simple classifiers like SVMs as in [Wang & Klaser$^+$ 11, Wang & Schmid 13]. With the success of deep convolutional networks (ConvNets) as image analysis models, in the past alot of research was devoted to extend ConvNets to process video data. However, with all the success of ConvNets as image analysis models, they suffer to achieve similar results in action recognition. Main difficulties arise because of limited training data, large intra class variation due to diverse view point, occlusions and backgrounds, and limited computing resources to model dependencies in longer video sequences. In action recognition, there are two complementary aspects: appearance of objects in individual frames and dependency across the frames. Hence, any good action recognition system should be able to extract and utilize both appearance and dependency. ConvNets are good at modelling appearance, however they suffer modelling dependency in longer sequences. To address the problem of modelling dependencies, there were number of approaches in the past. The earliest attempt to address action recognition by ConvNets is by feeding stack of consecutive frames as input to the network, and model is assumed to learn inter frame dependencies. However, the results were significantly worse than the results achieved by the hand crafted features. Other approaches like the two stream architecture [Simonyan & Zisserman 14a] decouple modelling appearance in individual frames and modelling dependency across the frames and finally fuse the output of two streams.

Usually, the temporal dependency in the sequence is modelled by recurrent neural networks (RNNs). In this work, we propose a recurrent ConvNet architecture called spatio temporal residual network, to model and utilize both appearance in individual frames and dependency across the frames. Our model is based on the state of the art image analysis residual learning framework [He & Zhang$^+$ 15]. In [He & Zhang$^+$ 15] a layer in the network is reformulated as learning a residual function with respect to the layer's input. With this reformulation, a layer can learn spatial residuals. We extend their work and propose the feature vector of frame at time $t$ to be a residual function with respect to the frame at time $t-1$.

The remainder of thesis is organized as follows: in Chapter 2, we introduce some related work. Technical details are given in Chapter 3. Spatio Temporal Residual Networks are explained in Chapter 4, in Chapter 5 we explain experiments and show empirical results, and finally Chapter 6 conclude our work.

# 2 Related Work

The task of action recognition in videos is mainly driven by the advances in image recognition methods. Hence, most of the action recognition methods are extensions of standard image recognition methods. Historically, the research work to address action recognition can be divided into three categories. The first category focuses on hand crafted features and bag of visual word representation. The second category replaces the bag of visual words representation with the Fisher vector representation. Finally, deep ConvNet architectures are explored for action recognition.

In the past, there were multiple approaches that rely on hand crafted features. In [Laptev & Marszalek$^+$ 08], each video is described by the bag of visual words. Sparse spatio temporal interest points are detected by the extension of Harris interest point detector. Each interest point is then described by the local spatio temporal features such as histogram of oriented gradients and histogram of optical flow. The features are then encoded by a bag of visual words representation and pooled over several spatio temporal grids. Finally, a SVM classifier is trained to classify the videos. The state of the art video feature representations make use of dense point trajectories. As in [Wang & Klaser$^+$ 11], each video is represented by dense point trajectories. Dense points from each frame are sampled and tracked based on displacement information from a dense optical flow field. A descriptor based on motion boundary histogram was introduced to encode the trajectory information. Finally, for evaluation, a standard bag of features approach is used. A codebook is constructed by fixing the number of visual words to 4000. For classification, a non-linear SVM is used. The approach of [Wang & Schmid 13] is also based on dense trajectories. However they improve dense trajectories features by taking into account the camera motion and removing all trajectories that are consistent with it. For evaluation, they used bag of features approach similar to [Wang & Klaser$^+$ 11] and Fisher vectors [Perronnin & Sánchez$^+$ 10]. Unlike bag of features, Fisher vectors encodes both first and second order statistics between the video descriptors and a Gaussian Mixture Model (GMM). It is also shown in [Ken Chatfield & Zisserman 11, Oneata & Verbeek$^+$ 13] that Fisher vector encoding outperforms standard bag visual feature encoding, both for image and for action recognition.

In the past, it had also been tried combine deep models and hand crafted features. [Peng & Zou$^+$ 14] proposed Stacked Fisher Vectors (SFV), a video representation with multi-layer nested Fisher vector encoding for action recognition. In the first layer, they densely sample large subvolumes from input videos, extract local features, and encode them using Fisher vectors. The second layer compresses the FVs of subvolumes obtained in the previous layer, and then encodes them again with Fisher vectors. Compared with standard FV, SFV allows refining the representation and abstracting semantic information in a hierarchical way. In [Jhuang & Serre$^+$ 07], an HMAX [Riesenhuber & Poggio 99] based architecture for action recognition was proposed with pre-defined spatio-temporal filters in the first layer.

In the past, there have been attempts to address the task of action recognition with deep architectures. However, in most of these works, the input to the model is a stack of consecutive video frames and the model is expected to learn spatio-temporal dependent features in the first few layers, which is a difficult task. In [Taylor & Fergus$^+$ 10, Chen & Ting$^+$ 10, Le & Zou$^+$ 11], spatio temporal features are learned in unsupervised fashion by using Restricted Boltzmann machines. The approach of [Jain & van Gemert$^+$ 15] combines the information about objects present in the video with the motion in the videos. 3D convolution is used in [Ji & Xu$^+$ 13] to extract discriminative spatio temporal features from the stacked of video frames. Three different approaches (early fusion, late fusion and slow fusion) were evaluated to fuse temporal context in [Karpathy & Toderici$^+$ 14]. A similar technique as in [Ji & Xu$^+$ 13] is used to fuse temporal context early in the network, in late fusion, individual features per frame are extracted and fused in the last convolutional layer. Slow fusion mixes late and early fusion. Trajectory pooled deep convolutional descriptors are defined in [Wang & Qiao$^+$ 15]. CNN features are extracted from the two stream architecture and are combined with improved dense trajectories. More recently, [Bilen & Fernando$^+$ 16] proposed concept of dynamic images. The dynamic image is based on the rank pooling concept [Fernando & Gavves$^+$ 15] and is obtained through the parameters of a ranking machine that encodes the temporal evolution of the frames of the video. Dynamic images are obtained by directly applying rank pooling on the raw image pixels of a video producing a single RGB image per video. And finally, by feeding the dynamic image to any ConvNet architecture for image analysis, it can be used to classify actions.

The algorithm of [Simonyan & Zisserman 14a] uses two convolutional networks (spatial and temporal networks). Individual frames from the videos are the input to the spatial network, while motion in the form of dense optical flow is the input to temporal network. The features learned by both networks are concatenated and finally linear SVM is used for classification. [Wang & Xiong$^+$ 15] followed the idea of [Simonyan & Zisserman 14a]. However, they modified the architecture of spatial and temporal networks. They replaced original network architectures with deep architectures. VGGNets [Simonyan & Zisserman 14b] are used as spatial and temporal network in two stream architecture. They also evaluated GoogleNet [Szegedy & Liu$^+$ 14] as replacement. Recently, with the success of ResNet [He & Zhang$^+$ 15], [Feichtenhofer & Pinz$^+$ 16] proposed a model that combines ResNet and the two stream architecture. They replace both spatial and temporal networks in two stream architecture by a ResNet with 50 layers. They also introduce temporal or motion residual, i.e. a residual connection from temporal network to the spatial network to all enable learning of spatio temporal features. [Wang & Xiong$^+$ 16] proposed the temporal segment networks, which are mainly based on the two stream architecture. However, rather than densely sampling every other frame in the video, they divided the video in $k$ segments of equal length, and then randomly sample $k$ snippets from corresponding segments, that forms the input for the two stream network. In this way, the two stream network produces segment level classification scores, which are combined to produce video level output.

Deep recurrent ConvNet architectures are also explored to model dependencies across the frames. In [Donahue & Hendricks$^+$ 14], convolutional layers are combined with LSTMs [Hochreiter & Schmidhuber 97] to model temporal dependencies. [Yang & Molchanov$^+$ 16] considered four networks to address action recognition in videos. The first network is similar to spatial network in the two stream architecture. The second network is a ConvNet with one recurrent layer, it expects a single optical flow image and in recurrent layer, optical flows over a range of frames are combined. In the third network, they feed a stack of consecutive frames, the network is also equipped with a recurrent layer to capture the long term dependencies. Similarly, the fourth similar to the temporal stream in the two stream architecture expects stack of a optical flow fields as input however, the network is equipped with fully connected recurrent layer. Finally, boosting is used to combine the output of all four networks.

# 3 Technical Details

This chapter illustrates the technical details. Overview of the neural networks is given in Section 3.1, and finally, different optimization techniques are discussed in Section 3.2.

## 3.1 Neural Networks

Neural networks are mathematical models, inspired by the biological nervous system, which enable a computer system to learn from experience. A neural network comprises a lot of small computing units, called neurons. Each neuron computes a linear transformation of the input, and then applies some non linear activation function. In practice sigmoid, ReLu, or sometimes identity is used as activation function. Neural networks learn by examples. There are numerous different types of neural networks, mainly the difference is in the way neurons are organized. Some of the most common neural networks are feed forward networks, convolutional networks and recurrent neural networks. Before defining each type of the network, we will introduce their basic components.

**Neuron**
A neuron is the basic computing unit in a neural network. It has a weighted incoming connection from every input. Within each neuron, a weighted combination of the input is computed and an activation function is applied to the weighted combination. Figure 3.1 illustrates a single neuron.

Mathematically, the function of a neuron $i$ in layer $l$, has two parts, i.e. to compute a pre-activation $O_i^{(l)}$ and an activation $A_i^{(l)}$. Finally $O_i^{(l)}$ and $A_i^{(l)}$ are computed as

$$O_i^{(l)} = W_i^{(l)} x + b_i^{(l)}, \tag{3.1}$$

$$A_i^{(l)} = \sigma\big(O_i^{(l)}\big), \tag{3.2}$$

where $\sigma(.)$ is an activation function, $x$ is the input vector and $W_i^{(l)}$ is a row vector of weights of incoming connections to the neuron $i$ in layer $l$, and $b_i^{(l)}$ is the bias.

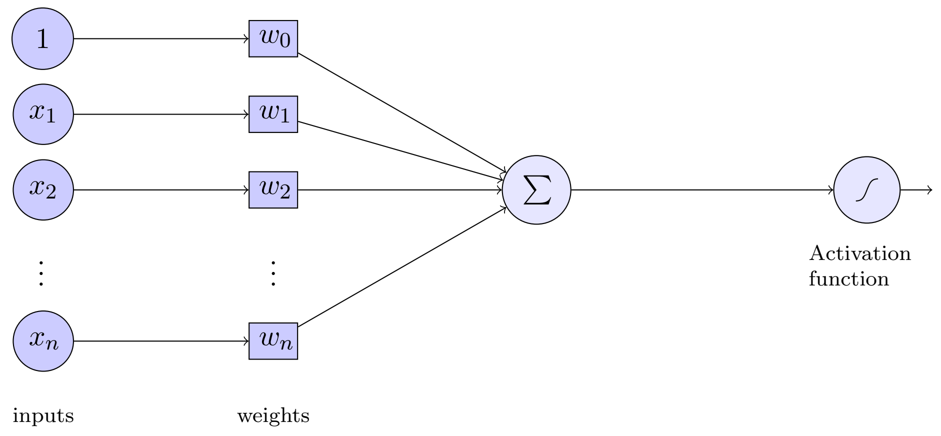


Figure 3.1: Single Neuron, each component of the input vector $x$ is weighted, the neuron first computes the weighted sum and then applies an activation function to it.

**Activation Function**
Activation functions are generally but not necessarily non linear functions. In general, they introduce non linarity to the network output. Non-linear output is to have the property of universal function approximation [Hornik & Stinchcombe$^+$ 89]. In practice, identity, sigmoid, or relu [Glorot & Bordes$^+$ 11] are used as activation functions for hidden layers and softmax is used as an activation function for the output layer.

**Identity**
Identity is a linear activation function, defined for all real numbers and has constant derivative at each point. For each input $x$, identity $\sigma(x)$ and its derivative are defined as

$$\sigma(x) = x,$$
$$\frac{d}{dx}\sigma(x) = 1.$$

Figure 3.2(a) plots identity activation function. In the context of neural networks it is an element wise activation function.

**Sigmoid**
Sigmoid is real valued, bounded differentiable function, defined for all real numbers, has a non negative derivative at each point and has a pair of horizontal asymptotes

i.e.

$$\lim_{x \to \infty} \sigma(x) = 1,$$
$$\lim_{x \to -\infty} \sigma(x) = 0.$$

Figure 3.2(b) plots the sigmoid activation function. Like identity, the sigmoid is also an element wise activation function. Formally, the sigmoid $\sigma(x)$ and its derivative, are defined as

$$\sigma(x) = \frac{1}{1+e^{-x}} \ ,$$
$$\frac{d}{dx}\sigma(x) = \sigma(x)\big(1 - \sigma(x)\big).$$

**Relu**
In the context of deep neural networks, the sigmoid activation function suffers from vanishing gradient problem. Vanishing gradient problem is a difficulty found in training deep networks with backpropogation. Due to backpropogation and small gradients of the sigmoid, the error signal decreases exponentially, hence initial layers in the network learn very slowly.

ReLu [Glorot & Bordes$^+$ 11] is a piece-wise linear function, defined for all real numbers. It is efficiently computable, scale invariant, has sparse activation for randomly initialized networks, and unlike sigmoid, ReLu doesn't suffer from the vanishing gradient problem. ReLu and its derivative are formally defined as

$$\sigma(x) = \begin{cases} x; & x > 0 \\ 0 & \end{cases}$$
$$\frac{d}{dx}\sigma(x) = \begin{cases} 1; & x > 0 \\ 0 & \end{cases}$$

Figure 3.2(c) plots ReLu. Like identity and sigmoid ReLu is also an element wise activation function.

**Softmax**
In practice, for classification problems, it is desirable to have a probability distribution as output of a neural network. The softmax function takes the input vector of arbitrary real values and map it to an output vector of real values, each in the range $[0, 1]$ that sums up to one. Hence, for classification, softmax is used as an activation function for last layer in the neural network. Let $P(y = c|x)$ be the probability that a given training example $x$ belongs to class $c$, then $P(y = c|x)$ is

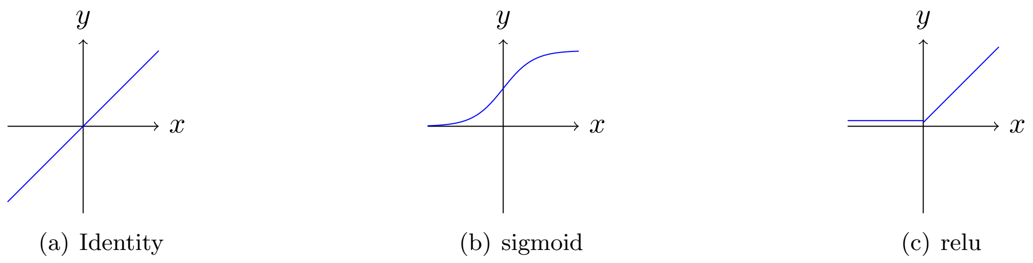


Figure 3.2: Activation Functions

calculated as

$$P(y=c|x) = \frac{e^{O_c^{(L)}}}{\sum_{\tilde{c}} e^{O_{\tilde{c}}^{(L)}}} \ ,$$

where $O^{(L)}$ is the pre-activation of output layer $L$ of the neural network, and $O_c^{(L)}$ is the pre-activation of a neuron $c$ in the output layer calculated by Equation (3.1). The derivative of the softmax function is defined as

$$\begin{aligned}\frac{\partial P(y=c|x)}{\partial O_k^{(L)}} &= \frac{\partial}{\partial O_k^{(L)}} \frac{e^{O_c^{(L)}}}{\sum_{\tilde{c}} e^{O_{\tilde{c}}^{(L)}}} \ , \\ &= \frac{\left(\sum_{\tilde{c}} e^{O_{\tilde{c}}^{(L)}}\right)\frac{\partial e^{O_c^{(L)}}}{\partial O_k^{(L)}} - e^{O_c^{(L)}} \frac{\partial \sum_{\tilde{c}} e^{O_{\tilde{c}}^{(L)}}}{\partial O_k^{(L)}}}{\left(\sum_{\tilde{c}} e^{O_{\tilde{c}}^{(L)}}\right)^2} \ .\end{aligned}$$

Here we distinguish two cases, as

Case $O_k^{(L)} = O_c^{(L)}$

$$\begin{aligned}\frac{\partial P(y=c|x)}{\partial O_k^{(L)}} &= \frac{\left(\sum_{\tilde{c}} e^{O_{\tilde{c}}^{(L)}}\right)e^{O_c^{(L)}} - e^{O_c^{(L)}} e^{O_c^{(L)}}}{\left(\sum_{\tilde{c}} e^{O_{\tilde{c}}^{(L)}}\right)^2} \ , \\ \frac{\partial P(y=c|x)}{\partial O_k^{(L)}} &= P(y=c|x)\big(1 - P(y=c|x)\big) \ . \qquad (3.3)\end{aligned}$$

Case $O_k^{(L)} \neq O_c^{(L)}$

$$\begin{aligned} \frac{\partial P(y=c|x)}{\partial O_k^{(L)}} &= \frac{-e^{O_c^{(L)}} e^{O_k^{(L)}}}{\left(\sum_{\tilde{c}} e^{O_{\tilde{c}}^{(L)}}\right)^2} \; , \\ \frac{\partial P(y=c|x)}{\partial O_k^{(L)}} &= -P(y=c|x)P(y=k|x) \; . \end{aligned} \tag{3.4}$$

**Objective Function**

Optimization methods minimize or maximize some objective function. Given some data and fixed parameters of an optimization method, the objective function rates how good the optimization method has performed on given data. For classification, cross entropy is used as objective function, and the parameters of the neural network are tuned to minimize the cross entropy loss. For a classification problem with $C$ classes and $N$ training examples, the cross entropy $J$ is defined as

$$J = -\frac{1}{N} \sum_n \sum_c Y_{n_c} log(P(y=c|x_n)) \; , \tag{3.5}$$

where $Y_n$ is one hot encoded vector, target for $nth$ training example.

## 3.1.1 Feed Forward Networks

In feed forward networks, neurons are organized in layers. There is a weighted connection from every neuron from layer $l$ to every neuron in layer $l+1$. Hence, the information is fed forward from one layer to the next. Input is fed into the first layer and output is observed at the last layer, the middle layers called hidden layers. Figure 3.3 depicts such a network.

Mathematically, a single layer neural network is capable of approximating any linear function. Organizing them in multiple layers enables universal function approximation, provided non-linear activation functions are used for atleast one hidden layer [Hornik & Stinchcombe$^+$ 89].

Analogous to a neuron, function of layer $l$ can be divided in two parts i.e. to compute pre-activation $O^{(l)}$ and activation $A^{(l)}$. Hence, equation (3.1) and (3.2) can be modified to compute layer's pre-activation and activation as

$$\begin{aligned} O^{(l)} &= W^{(l)}x \; + \; b^{(l)} \; , \\ A^{(l)} &= \sigma\big(O^{(l)}\big) \; , \end{aligned} \tag{3.7}$$

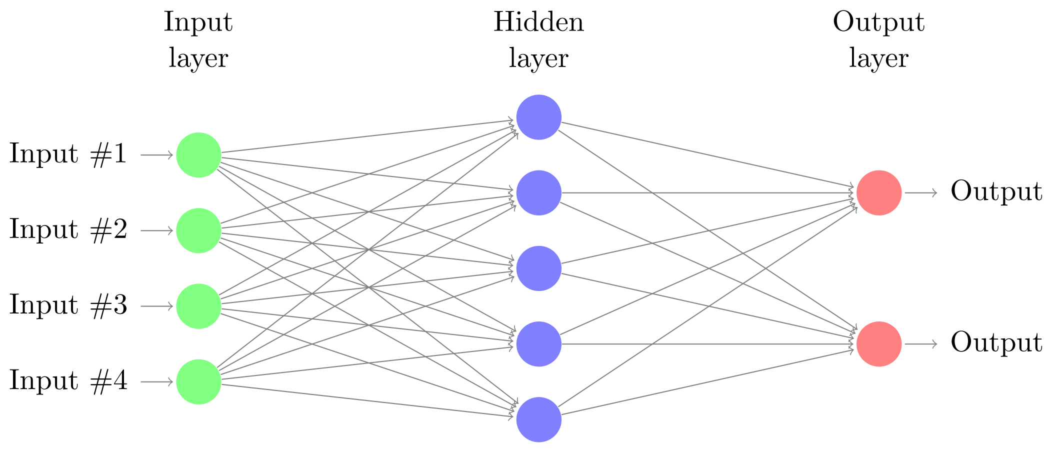


Figure 3.3: Feed Forward Network, network expects four dimensional input and produces two dimensional output.

Here, $x \in \mathbb{R}^d$ is a $d$ dimensional vector (input to layer $l$), $b^{(l)} \in \mathbb{R}^m$ is bias vector for layer $l$, and $W^{(l)} \in \mathbb{R}^{m \times d}$ is a weight matrix, between layer $l-1$ with $d$ units and layer $l$ with $m$ units.

Feed forward neural networks are family of functions $f : \mathbb{R}^d \rightarrow \mathbb{R}^m$. The output of a feed forward neural network is the activation of last layer called output layer. Function of a feed forward neural network with $L$ layers is defined as

$$\begin{aligned} f(x) &= A^{(L)} \ , \\ &= \sigma_L(O^{(L)}) \ , \\ &= \sigma_L(W^{(L)}A^{(L-1)} + b^{(L)}) \ , \\ f(x) &= \sigma_L(W^{(L)}\sigma_{L-1}(W^{(L-1)}...\sigma_1(W^{(1)}x + b^{(1)}) + ... + b^{(L-1)}) + b^{(L)}) \ . \end{aligned}$$

Here, $b^{(l)}$ is bias vector for layer $l$, $W^{(l)}$ is weight matrix between layer $l-1$ and layer $l$, $\sigma_l(.)$ is the activation function used for layer $l$, $O^{(l)}$ is vector of pre-activations and $A^{(l)}$ is vector of activations.

**Backpropagation Algorithm**

The backpropagation algorithm [Rumelhart & Hinton$^+$ 88] is the common way to train neural networks in combination with gradient descent as optimization method. Gradient descent is a first order iterative optimization algorithm.

Given a function $f_\theta$ with parameters $\theta$, to find the minimum of $f_\theta$, at each iteration the gradient descent updates $\theta$, proportional to negative gradient of $f_\theta$ with respect to $\theta$. In case of neural networks $\theta = \cup\{\theta^{(l)}\}_{l=1}^{L}$, where $\theta^{(l)}$ represents the parameters of a layer $l$, and $\theta^{(l)} = \{W^{(l)}, b^{(l)}\}$.

The backpropogation algorithm has two parts. It first calculates the output of the network in the forward pass. And in the backward pass, it calculates the error signals with respect to all parameters of the network. It calculates error at the output layer by comparing the desired output and the output produced by the network, calculates error signals with respect to parameters of output layer, recursively backward propagate the error through all layers by application of the chain rule and calculates the error signals with respect to parameters of the respective layers. Algorithm 1 outlines pseudo code of the backpropagation algorithm.

**Algorithm 1** Backpropagation Algorithm

```
1: procedure BACKPROPAGATION
2:     initialize $\{\theta^{(l)}\}_{l=1}^{L}$ randomly
3:     while termination do
4:         initialize $\{\Delta\theta^{(l)}\}_{l=1}^{L}$ to zero
5:         for each training example $x$ do
6:             $prediction \leftarrow forwardPass(x)$
7:             $target \leftarrow desiredOutput(x)$
8:             $\delta^{(L)} \leftarrow computeError(prediction, target)$
9:             $\nabla\theta^{(L)} \leftarrow computeGradient(\delta^{(L)}, \theta^{(L)})$
10:            $\Delta\theta^{(L)} \leftarrow \Delta\theta^{(L)} + \nabla\theta^{(L)}$
11:            for $l = L - 1$ to 1 do
12:                $\delta^{(l)} \leftarrow backPropagate(\delta^{(l+1)}, \theta^{(l+1)})$
13:                $\nabla\theta^{(l)} \leftarrow computeGradient(\delta^{(l)}, \theta^{(l)})$
14:                $\Delta\theta^{(l)} \leftarrow \Delta\theta^{(l)} + \nabla\theta^{(l)}$
15:            end
16:        end
17:        for each layer $l$ do
18:            $\theta^{(l)} \leftarrow \theta^{(l)} - learningRate \cdot \Delta\theta^{(l)}$
19:        end
20:    end
```

In the following, we explain, how the backpropagation algorithm backpropagates the error signal and how it calculates the gradient with respect to each parameter. The error signal or gradient of loss $J$ with respect to each parameter is calculated by the application of chain rule. For instance error signal or gradient of $J$ with respect to weight matrix $W^{(l)}$ at a layer $l$ is computed as

$$\begin{aligned}\frac{\partial J}{\partial W^{(l)}} &= \frac{\partial J}{\partial O^{(l)}}\frac{\partial O^{(l)}}{\partial W^{(l)}},\\ &= \frac{\partial J}{\partial A^{(l)}}\frac{\partial A^{(l)}}{\partial O^{(l)}}\frac{\partial O^{(l)}}{\partial W^{(l)}},\\ \frac{\partial J}{\partial W^{(l)}} &= \frac{\partial J}{\partial O^{(l+1)}}\frac{\partial O^{(l+1)}}{\partial A^{(l)}}\frac{\partial A^{(l)}}{\partial O^{(l)}}\frac{\partial O^{(l)}}{\partial W^{(l)}}.\end{aligned} \tag{3.8}$$

It is evident from Equation (3.8), that the term $\frac{\partial J}{\partial O^{(l)}}$ will reappear in the gradient, with respect to the parameters of the preceding layer. We call the term $\frac{\partial J}{\partial O^{(l)}}$, the error $\delta^{(l)}$ at layer $l$. The backpropagation algorithm backpropagates $\delta^{(l)}$ while calculating the gradient in the preceding layers. Hence, the error $\delta^{(l)}$ for a layer $l$ is defined as

$$\delta^{(l)} = \frac{\partial J}{\partial O^{(l)}}.$$

Here, $J$ is an objective function and $O^{(l)}$ is a vector of pre-activations for layer $l$. For cross entropy, Equation (3.5), as objective function and $(x, Y)$ as training example, the error $\delta_i^{(L)}$ at the neuron $i$ in the output layer $L$ is calculated as

$$\begin{aligned}\delta_i^{(L)} = \frac{\partial J}{\partial O_i^{(L)}} &= -\frac{\partial}{\partial O_i^{(L)}}\sum_k Y_k log(P(y=k|x)),\\ &= -\sum_k Y_k \frac{1}{P(y=k|x)}\frac{\partial P(y=k|x)}{\partial O_i^{(L)}}.\end{aligned}$$

Using Equation (3.3) and Equation (3.4), we obtain

$$\begin{aligned}\frac{\partial J}{\partial O_i^{(L)}} &= -\big(Y_i(1-P(y=i|x)) - \sum_{k\neq i} Y_k P(y=k|x)\big),\\ &= -\big(Y_i - P(y=i|x)\big(\sum_k Y_k\big)\big).\end{aligned}$$

Using the fact, that $\sum_k Y_k = 1$, further simplifications leads to

$$\frac{\partial J}{\partial O_i^{(L)}} = -\big(Y_i - P(y=i|x)\big).$$

Finally, the error vector $\frac{\partial J}{\partial O^{(L)}}$, containing the error for each neuron in the output layer $L$ is

$$\frac{\partial J}{\partial O^{(L)}} = -(Y - A^{(L)}) = \delta^{(L)} \ , \tag{3.10}$$

where $Y$ is a one hot encoded vector and $A^{(L)}$ is vector of activations for the output layer. Similarly, the error $\delta_i^{(l)}$ for a neuron $i$ in the hidden layer $l$ can be computed by applying chain rule as

$$\delta_i^{(l)} = \frac{\partial J}{\partial O_i^l} = \frac{\partial J}{\partial A_i^{(l)}} \frac{\partial A_i^{(l)}}{\partial O_i^{(l)}}. \tag{3.11}$$

Using Equation (3.7), we obtain

$$\frac{\partial A_i^{(l)}}{\partial O_i^{(l)}} = \sigma'(O_i^{(l)}). \tag{3.12}$$

For $\frac{\partial J}{\partial A_i^{(l)}}$, further application of the chain rule results in

$$\begin{aligned} \frac{\partial J}{\partial A_i^{(l)}} &= \sum_j \frac{\partial J}{\partial O_j^{(l+1)}} \frac{\partial O_j^{(l+1)}}{\partial A_i^{(l)}}, \\ &= \sum_j \delta_j^{(l+1)} \frac{\partial \sum_k A_k^{(l)} W_{jk}^{(l+1)}}{\partial A_i^{(l)}}, \\ &= \sum_j \delta_j^{(l+1)} W_{ji}^{(l+1)}. \end{aligned} \tag{3.13}$$

Substituting Equation (3.12) and Equation (3.13) into (3.11), we get

$$\begin{aligned} \frac{\partial J}{\partial O_i^{(l)}} &= \big(\sum_j \delta_j^{(l+1)} W_{ji}^{(l+1)}\big) \sigma'(O_i^{(l)}) \ , \\ &= \big(W_i^{(l+1)^T} \delta^{(l+1)}\big) \sigma'(O_i^{(l)}) \ . \end{aligned}$$

Hence, the error vector $\delta^{(l)}$, containing the error for each neuron in hidden layer $l$ is calculated as

$$\frac{\partial J}{\partial O^{(l)}} = \big(W^{(l+1)^T} \delta^{(l+1)}\big) \odot \sigma'(O^{(l)}) = \delta^{(l)} \ , \tag{3.14}$$

where $\odot$ represents the element-wise vector product. Finally gradient of $J$ with respect parameters of layer $l$, $W^{(l)}$ and $b^{(l)}$ are

$$\frac{\partial J}{\partial W^{(l)}} = \frac{\partial J}{\partial O^{(l)}} \frac{\partial O^{(l)}}{\partial W^{(l)}},$$
$$\frac{\partial J}{\partial b^{(l)}} = \frac{\partial J}{\partial O^{(l)}} \frac{\partial O^{(l)}}{\partial b^{(l)}}.$$

Replacing $\frac{\partial J}{\partial O^{(l)}}$ by $\delta^{(l)}$, we get

$$\frac{\partial J}{\partial W^{(l)}} = \delta^{(l)} A^{(l-1)T}, \tag{3.15}$$
$$\frac{\partial J}{\partial b^{(l)}} = \delta^{(l)}. \tag{3.16}$$

So, in case of feed forward networks, Equation (3.10) is used in step 8 of the back-propagation algorithm, to calculate the error signal at the output layer, Equation (3.14) is used in the step 11 to calculate the error signal for hidden layers and finally Equations (3.15) and (3.16) are used to calculate the gradients with respect to the parameters.

### 3.1.2 Convolutional Neural Networks

One problem that arises in applying feed forward neural networks to solve any of the computer vision problem, is the way neurons are connected in feed forward networks. Because of the fact, adjacent layers are fully connected, applying such a network, results in a huge number of parameters. Hence, it would be very expensive to compute linear transformations, and huge amount of data would be required to get a good generalization. Convolutional neural networks, also called ConvNets, are a biologically inspired variant of feed forward networks. These kind of networks have shown tremendous performance in computer vision problems.

ConvNets are similar to feed forward networks. However, the connectivity patterns of neurons in ConvNets is inspired by the organization of the animal visual cortex [Hubel & Wiesel 68]. The animal visual cortex contains a complex arrangement of cells. These cells respond to small subregions of the visual field, called receptive field. This phenomena in ConvNets is achieved by local connectivity, i.e. connecting neurons to small subregions of the input. With local connections between the neurons of adjacent layers, ConvNets exploit spatially local correlations.

ConvNets segment neurons of a layer into feature maps. One feature map cov-

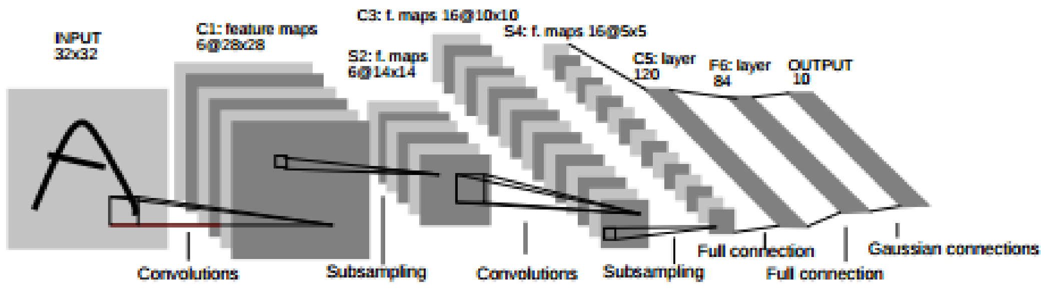


Figure 3.4: Architecture of LeNet-5 [LeCun & Haffner+ 99], a ConvNet for digit recognition.

ers whole input area and all the neurons within one feature map share parameters. Parameter sharing is another noticeable difference between ConvNets and feed forward networks. Parameter sharing further reduces the model size. Also, by parameter sharing, ConvNets achieve translation invariance.

Convolutional layers and pooling or subsampling layers are two main ingredients of ConvNets. ConvNets contain one or more convolutional and pooling layers followed by fully connected layers as in feed forward networks. Figure 3.4 shows the architecture of a ConvNet. In Figure 3.4, ConvNet outputs six feature maps in first convolutional layer, second layer is pooling layer, third layer outputs sixteen feature maps, forth layer is again pooling layer, finally there are three fully connected layers. In the following we will explain the convolutional and the pooling layers, and how error is backpropagated through such layers.

**Convolutional Layer**
Neurons in the convolutional layer are sparsely connected to neurons in the source layer. Pre-activations in such a layer are calculated by discrete convolution rather than matrix multiplication.

**Convolution** is a linear and shift invariant operation that blends one function with another. In case of computer vision, convolution blends an image $I$ with kernel $K$. Discrete convolution is defined as

$$(I * K)[x, y] = \sum_{a} \sum_{b} K[a, b] \cdot I[x - a, y - b] \ ,$$

where $I$ and $K$ are the image and the kernel, respectively. Let $(0, 0)$ be the center of the kernel $K$, and summation indices $a, b$ iterate from $(-\frac{K.w}{2}, -\frac{K.h}{2})$ to $(\frac{K.w}{2}, \frac{K.h}{2})$,

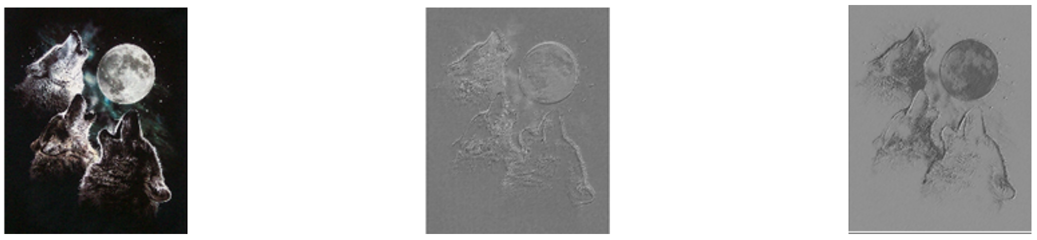


Figure 3.5: Convolutional Layer output feature maps. Image taken from [deeplearning.net]

with $K.w$ and $K.h$ being the width and the height of the kernel $K$. The size of the kernel represents the size of the receptive field.

**Correlation** is another linear shift invariant operation, i.e. just like convolution with the exception that the kernel is flipped in both dimensions. In practice, for convolutional layers, either correlation or convolution is used as an operation to calculate pre-activations. Discrete correlation is defined as

$$(I \circ K)[x, y] = \sum_a \sum_b K[a, b] \cdot I[x + a, y + b].$$

The input to a convolutional layer is an image of size $W \times H \times C$, where $W$ and $H$ represents the height and the width of the image and $C$ represents number of channels. A convolutional layer has $D \times C$ filters, each of size $w \times h$ to produce $D$ feature maps. Figure 3.5 shows for an input image (a), two output feature maps (b) and (c) of a convolutional layer. It is evident from Figure 3.5 that the convolutional layers also learn filters similar to hand crafted features like gradients. Pre-activation $O_d^{(l)}$ in convolutional layer $l$ for feature map $d$ is calculated as

$$O_d^{(l)} = \sum_c A_c^{(l-1)} * K_{(c,d)}^{(l)} + b_d^{(l)}. \tag{3.17}$$

The activations $A_d^{(l)}$ for a feature map $d$ in a convolutional layer $l$ are calculated as

$$A_d^{(h)} = \sigma(O_d^{(h)}). \tag{3.18}$$

In order to apply the backpropagation algorithm to learn the parameters of a convolutional layer, we need to define the error signal a of convolutional layer and gradient with respect to the bias and the kernels. Analogous to feed forward networks, the error $\delta^{(l)}_{c(x,y)}$ for layer $l$, feature map $c$ at pixel $(x,y)$, is calculated as

$$\delta^{(l)}_{c(x,y)} = \frac{\partial J}{\partial O^{(l)}_{c(x,y)}}. \tag{3.19}$$

By applying the chain rule, we get

$$\delta^{(l)}_{c(x,y)} = \frac{\partial J}{\partial A^{(l)}_{c(x,y)}} \frac{\partial A^{(l)}_{c(x,y)}}{\partial O^{(l)}_{c(x,y)}}. \tag{3.20}$$

As each input feature contributes to all output features, and a pixel $(x,y)$ contributes to all receptive fields indexed by $x', y'$, it follows that

$$\frac{\partial J}{\partial A^{(l)}_{c(x,y)}} = \sum_d \sum_{x'} \sum_{y'} \frac{\partial J}{\partial O^{(l+1)}_{d(x',y')}} \frac{\partial O^{(l+1)}_{d(x',y')}}{\partial A^{(l)}_{c(x,y)}}.$$

Using Equation (3.19) and Equation (3.17), we get

$$\frac{\partial J}{\partial A^{(l)}_{c(x,y)}} = \sum_d \sum_{x'} \sum_{y'} \delta^{(l+1)}_{d(x',y')} \frac{\partial}{\partial A^{(l)}_{c(x,y)}} \sum_e \sum_a \sum_b K^{(l+1)}_{e,d(a,b)} \cdot A^{(l)}_{e(x'-a,y'-b)} + b^{(l+1)}_{d(x',y')}.$$

Using the fact that all feature maps $A^{(l)}_e$ are constants with respect to $A^{(l)}_c$ except when $A^{(l)}_e = A^{(l)}_c$, and $A^{(l)}_{c(x'-a,y'-b)}$ is constant with respect to $A^{(l)}_{c(x,y)}$ except when $x' - a = x$ and $y' - b = y$, it follows

$$\frac{\partial J}{\partial A^{(l)}_{c(x,y)}} = \sum_d \sum_{x'} \sum_{y'} \delta^{(l+1)}_{d(x',y')} \cdot K^{(l+1)}_{c,d(x'-x,y'-y)}.$$

It follows from the definition of convolution

$$\begin{aligned}\frac{\partial J}{\partial A^{(l)}_{c(x,y)}} &= \sum_d \delta^{(l+1)}_{d(x,y)} * K^{(l+1)}_{c,d(-x,-y)}, \\ &= \sum_d \delta^{(l+1)}_{d(x,y)} * flip\big(K^{(l+1)}_{c,d(x,y)}\big).\end{aligned}$$

By the definition of correlation, we get

$$\frac{\partial J}{\partial A^{(l)}_{c(x,y)}} = \sum_d \delta^{(l+1)}_{d(x,y)} \circ K^{(l+1)}_{c,d(x,y)}. \tag{3.21}$$

By using Equation (3.18), it follows

$$\frac{\partial A^{(l)}_{c(x,y)}}{\partial O^{(l)}_{c(x,y)}} = \sigma'\big(O^{(l)}_{c(x,y)}\big). \tag{3.22}$$

By substituting Equation (3.21) and Equation (3.22) into Equation (3.20), we get

$$\delta^{(l)}_{c(x,y)} = \Big(\sum_d \delta^{(l+1)}_{d(x,y)} \circ K^{(l+1)}_{c,d(x,y)}\Big)\sigma'\big(O^{(l)}_{c(x,y)}\big).$$

Finally, the error signal $\delta^{(l)}_c$ at convolutional layer $l$ for feature map $c$ is given by

$$\delta^{(l)}_c = \Big(\sum_d \delta^{(l+1)}_d \circ K^{(l+1)}_{(c,d)}\Big) \odot \sigma'(O^{(l)}_c). \tag{3.23}$$

Finally, it can be shown that

$$\frac{\partial J}{\partial K^{(l)}_{(b,c)}} = \delta^{(l)}_c \circ A^{(l-1)}_b, \tag{3.24}$$

$$\frac{\partial J}{\partial b^{(l)}_c} = \delta^{(l)}_c. \tag{3.25}$$

So, Equation (3.23) is used to compute the error signal for a convolutional layer in the backpropagation algorithm, Equations (3.24) and (3.25) are used to compute the gradients with respect to parameters of convolutional layer.

**Pooling Layer**
Pooling layers progressively reduce the spatial size of the input to reduce the number of parameters, computations in the network and the effect of overfitting. Pooling layers operate independently on each input feature map. The input to a pooling layer is a $W \times H \times C$ image, where $C$ is the number of input feature maps and $W, H$ are the width and the height of each feature map, respectively. The output of a pooling layer is $W_o \times H_o \times C$, where $W_o, H_o$ are the width and the height of each output feature map. $W_o$ and $H_o$ are calculated as

$$W_o = \lceil W/S \rceil \; and$$
$$H_o = \lceil H/S \rceil \;,$$

where $S$ is the stride. Max pooling and average pooling, among others, are the two most commonly used pooling layer types.

**Max Pooling** is a non linear down sampling operation. Given the input feature map $I_c$ and the pooling window $P$ of size $G$ starting at $(x, y)$, max pooling is computed as

$$O_{(x/S,y/S)} = max\{v \in P\}.$$

The error signal for $\delta^{(l)}$ for the max pooling layer $l$, is computed by computing the error signal for each window of size $P$. The error signal $\delta^{(l)}_{(x,y)}$ for a window $P$ starting at $(x, y)$ is calculated by creating a window $P_b$ of zeros of size $G$, and defining $P_b[i, j] = \delta^{(l+1)}_{(x/S,y/S)}$, if in the forward pass $ij$ was the index of the maximum.

**Average Pooling** is a linear down sampling operation. Given the input feature map $I$ and a pooling window $P$ of size $G$ starting at $(x, y)$, average pooling is defined as

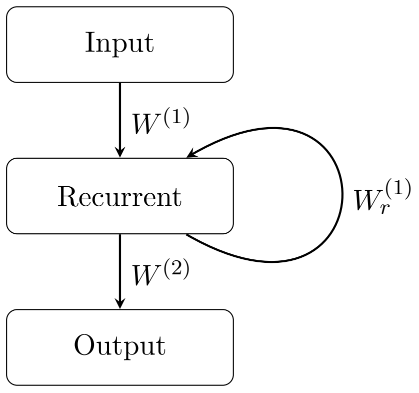
$$O_{(x/S,y/S)} = \frac{1}{G}\sum_i\sum_j v_{ij} \ .$$

Analogous to the max pooling layer, the error signal $\delta^{(l)}$ of the average pooling layer $l$ is also computed by computing the error signal $\delta^{(l)}_{(x,y)}$ for each window $P$, starting at $(x, y)$. For computing the error signal for a window $P$, a window $P_b$ of size $G$ is created, and finally $P_b[i, j] = \frac{1}{G}\delta^{(l+1)}_{(x/S,y/S)}$, for all $i, j \in P$.

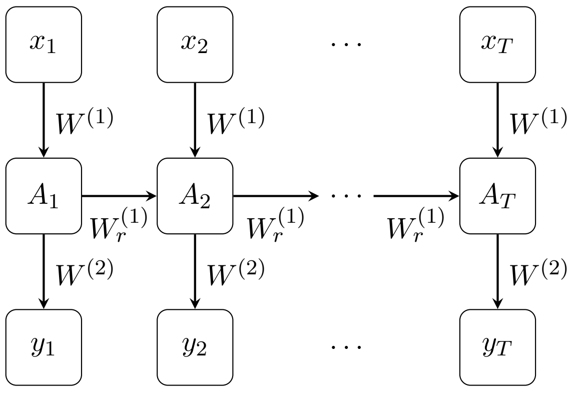
### 3.1.3 Recurrent Networks

Both ConvNets and feed forward networks try to find a mapping between $x$ and $y$, i.e. both ConvNets and feed forward networks assume that output $y$ depends on a single input $x$. However, there are problems in which the output does not depend on single input but rather on a sequence of inputs, hence there is a need to model sequences. In order to do sequence modelling, both ConvNets or feed forward neural networks are modified to have one or more recurrent layers. Such networks are called recurrent neural networks or RNNs.

Mathematically, RNNs are more powerful than feed forward networks and ConvNets. It is shown in [Siegelmann & Sontag 92] that RNNs are Turing complete. RNNs try to find a mapping between sequence of inputs $\{x\}_{t=1}^{T}$ to a single output $y$ or sequence of outputs $\{y\}_{t=1}^{T}$. There are numerous applications of RNNs, for instance, action recognition, language modelling, image description generation, and machine translation. Figure 3.6(a) shows a Recurrent Neural Network and 3.6(b) shows a RNN unfolded through time.



(a) Recurrent Network



(b) RNN unfolded

Figure 3.6: Recurrent Neural Network.

Recurrent layers in the recurrent networks, act as an internal memory. They store contexual information. The output of a recurrent layer not only depends on the input at time step $t$, but also on the output of the recurrent layer at time step $t-1$. Parameters of recurrent networks are shared for all time steps. The pre-activations $O_t^{(l)}$ for fully connected recurrent layer $l$ at time step $t$ are calculated as

$$O_t^{(l)} = W^{(l)} A_t^{(l-1)} + W_r^{(l)} A_{t-1}^{(l)} + b^{(l)}.$$

Here, $W^{(l)}$ represents the weight matrix containing weights of the connections from layer $l-1$ to the recurrent layer $l$ and $W_r^{(l)}$ represents the weight matrix of the recurrent connection. Finally, the activations $A_t^{(l)}$ for recurrent layer $l$ at time step $t$ are calculated similar to feed forward networks, i.e.

$$A_t^{(l)} = \sigma(O_t^{(l)}).$$

Similarly, for a recurrent convolutional layer $l$, the pre-activations $O_{t_d}^{(l)}$ at time step $t$ for feature map $d$ are calculated as

$$O_{t_d}^{(l)} = \sum_c A_{t_c}^{(l-1)} * K_{c,d}^{(l)} + \sum_e A_{(t-1)_e}^{(l)} * K_{r_{e,d}}^{(l)} + b_d^{(l)} \ ,$$

where $A_{t_c}^{(l-1)}$ represents the activations from layer $l-1$ for feature map $c$ at time step $t$, $A_{(t-1)_e}^{(l)}$ represents the activations from the recurrent layer $l$ for feature map $e$ at time step $t-1$, $K_{c,d}^{(l)}$ represents the convolutional weights, $K_{r_{e,d}}^{(h)}$ represents the recurrent convolutional weights, and $b_d^{(h)}$ represents the bias.

RNNs are trained with backpropagation through time. It works analogous to the standard backpropagation algorithm with the exception that gradients with respect to recurrent weights are calculated differently:

$$\delta^{(l)} = \frac{\partial J}{\partial O_T^{(l)}} = \big(W^{(l+1)^T}\delta^{(l+1)}\big) \odot \sigma'\big(O_T^{(l)}\big).$$

The gradient of the loss with respect to recurrent weights $W_r^{(l)}$ at layer $l$ is then calculated as

$$\frac{\partial J}{\partial W_r^{(l)}} = \frac{\partial J}{\partial O_T^{(l)}} \sum_{t=1}^{T} \frac{\partial O_T^{(l)}}{\partial O_t^{(l)}} \frac{\partial O_t^{(l)}}{\partial W_r^{(l)}} \ . \tag{3.29}$$

By applying the chain rule, we get

$$\frac{\partial O_T^{(l)}}{\partial O_t^{(l)}} = \frac{\partial O_T^{(l)}}{\partial O_{T-1}^{(l)}} \frac{\partial O_{T-1}^{(l)}}{\partial O_{T-2}^{(l)}} \cdots \frac{\partial O_{t+1}^{(l)}}{\partial O_t^{(l)}} \ . \tag{3.30}$$

Equation (3.30) can be simplified by defining the error $\delta_t^{(l)}$ for the recurrent layer $l$ at time step $t$ as

$$\delta_t^{(l)} = \delta_{t+1}^{(l)} \frac{\partial O_{t+1}^{(l)}}{\partial O_t^{(l)}}. \tag{3.31}$$

Substituting Equation (3.31) into Equation (3.29), we get

$$\frac{\partial J}{\partial W_r^{(l)}} = \frac{\partial J}{\partial O_T^{(l)}} \sum_{t=1}^{T} \delta_t^{(l)} \frac{\partial O_t^{(l)}}{\partial W_r^{(l)}}. \tag{3.32}$$

With all the mathematical power the RNNs have, unfortunately they are incredibly hard to train. As it has been observed by [Bengio & Simard$^+$ 94], it is difficult to train RNNs to capture long-term dependencies because gradients either tend to vanish, or sometimes explode. It is also evident in Equations (3.32) and (3.30) that due to backpropagation through time, if the sequence is very long, the gradient can easily vanish or explode. The problem becomes more severe if the recurrent network is of significant depth. This makes gradient based optimization methods struggle, not just because of variation in gradient magnitude but because of the effect of long term dependencies is hidden by the effect of short term dependencies.

In the past, many researchers tried to reduce this negative impact. There were two dominant approaches i.e. to design better learning algorithm, as [Bengio & Boulanger-Lewandowski$^+$ 12] used clipped gradient, by which the norm of the gradient vector is clipped. The other approach is to design a sophisticated activation function i.e. linear transformation followed by element wise activation using gated units. The earliest attempt to design the sophisticated activation function is by [Hochreiter & Schmidhuber 97], called long term short term memory (LSTM). More recently [Cho & van Merrienboer$^+$ 14] proposed a gated recurrent unit, which is also a sophisticated activation function to capture long term dependencies.

## 3.2 Optimization Methods

The goal of an optimization method is to find parameters of a function $f$, for which function $f$ evaluates to minimum or maximum. In case of machine learning or neural networks, function $f$ refers to the loss function. Different optimization techniques exists, among them gradient based methods are the most common methods to tune parameters of neural networks.

Let $f_\theta$ be a function parametrized with vector of parameters $\theta$. To optimize $f_\theta$ with gradient based methods, gradient of $f_\theta$ with respect to every parameter is calculated and the corresponding parameter is changed in the positive or the negative direction of the gradient. In case of neural networks we always try to minimize some loss function, hence parameters are changed in the negative direction of the gradient. Such gradient based methods are called gradient descent. In this section two variants of gradient descent are introduced, some of the challenges they face are discussed and finally some of the most common methods addressing these challenges are discussed.

### 3.2.1 Batch Gradient Descent

In batch gradient descent, the gradient vector of the loss function is calculated for the whole training set. The updates to the parameters is performed once after the complete training set. Batch gradient descent tends to converge very well to the local minimum. As parameters are updated once after the complete training set, batch gradient descent is very slow, some times intractable, for large training sets. And also it is not easy to incorporate new data in an online setting.

### 3.2.2 Stochastic Gradient Descent

The main difference between stochastic gradient descent and batch gradient descent is, stochastic gradient descent updates parameters after each random mini batch. In this way, the method can easily be applied to an online setting, also the problem with huge training sets can be solved easily. However, as parameters are updated are updated based on stochastic approximation rather than based on actual gradient, the loss function fluctuates alot during training.

Both batch and stochastic gradient descent, update parameters by taking a step proportional to negative gradient of the loss function. The step size is determined by a hyper parameter called learning rate. In order to achieve good convergence, careful tuning of learning rate is very important and is often very difficult. A small learning rate leads to very slow convergence, however large learning rates can hinder the convergence and cause the loss function to fluctuate around a local minimum and sometimes can cause divergence.

### 3.2.3 RPROP

RPROP stands for "resilient propagation" and is proposed by [Riedmiller & Braun 93]. It is an adaptive batch learning algorithm. It adapts local learning rate based on local gradient information. In contrast to other adaptive methods, it only considers the sign of the gradient in each step. In RPROP, individual updates for each weight $w_{ij}$ at iteration $k$ is calculated as

$$\Delta w_{ij}^{k} = \begin{cases} -\Delta_{ij}^{k}, & if\ \frac{\partial J}{\partial w_{ij}^{k}} > 0 \\ +\Delta_{ij}^{k}, & if\ \frac{\partial J}{\partial w_{ij}^{k}} < 0 \\ 0, & else\ , \end{cases}$$

where, $\Delta_{ij}^{k}$ are calculated as

$$\Delta_{ij}^{k} = \begin{cases} \eta^{+} \cdot \Delta_{ij}^{k-1}, & if\ \frac{\partial J}{\partial w_{ij}^{k-1}} \cdot \frac{\partial J}{\partial w_{ij}^{k}} > 0 \\ \eta^{-} \cdot \Delta_{ij}^{k-1}, & if\ \frac{\partial J}{\partial w_{ij}^{k-1}} \cdot \frac{\partial J}{\partial w_{ij}^{k}} < 0 \\ \Delta_{ij}^{k-1}, & else. \end{cases}$$

Here, $\eta^-$ and $\eta^+$ are learning rates, that are usually set to $\eta^- = 0.5$ and $\eta^+ = 1.2$. Finally, weights are updated using the update rule

$$w_{ij}^{k+1} = w_{ij}^k + \Delta w_{ij}^k.$$

With this update rule, the magnitude of the update is increased if in successive iterations sign of the gradient stays same. The magnitude of the update is decreased if the gradient changes its sign between successive iterations.

### 3.2.4 ADAM

ADAM has recently been proposed by [Kingma & Ba 14], and is an adaptive learning method. It computes adaptive learning rates for different weights based on estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients. The method combines the advantages of AdaGrad [Duchi & Hazan$^+$ 11], which works well with sparse gradients, and RMSProp [Tieleman & Hinton 12], which works well in on-line and non stationary settings. ADAM works by calculating the gradient with respect to each parameter. Let $g_{ij}^k$ be the gradient of loss at iteration $k$, with respect to parameter $w_{ij}$ at iteration $k-1$, so

$$g_{ij}^k = \frac{\partial J}{\partial w_{ij}^{k-1}}.$$

The moments are estimated as

$$\begin{aligned} m_{ij}^k &= \beta_1 m_{ij}^{k-1} + (1-\beta_1) g_{ij}^k \ , \\ v_{ij}^k &= \beta_2 v_{ij}^{k-1} + (1-\beta_2) \big(g_{ij}^k\big)^2 \ , \end{aligned}$$

where $\beta_1, \beta_2 \in [0, 1)$ are the exponential decaying rates for the moment estimates. Both moments are initialized with zero, due to this the authors observed the moments are biased towards zero during initial steps. Hence, the bias corrected first moment $\mu_{ij}^k$ and second moment $\nu_{ij}^k$ are calculated as

$$\begin{aligned} \mu_{ij}^k &= \frac{m_{ij}^k}{(1-\beta_1^k)} \ , \\ \nu_{ij}^k &= \frac{v_{ij}^k}{(1-\beta_2^k)} \ , \end{aligned}$$

where $\beta_1^k, \beta_2^k$ are $\beta_1, \beta_2$ raised to power $k$ respectively. Finally parameters are updated using update rule

$$w_{ij}^k = w_{ij}^{k-1} - \eta \frac{\mu_{ij}^k}{(\sqrt{\nu_{ij}^k} + \epsilon)}.$$

Here, $\eta$ represents the learning rate, and epsilon is a small number to avoid division by zero.

### 3.2.5 Batch Normalization

Another complication in training the deep networks is, the input distribution of each layer changes during training, because it depends on the parameters of preceding layer. This phenomena is called internal covariate shift [Shimodaira 00]. Due to internal covariate shift, the process of training deep networks tremendously slows down, because learning rate needs to be very small. To cope with this problem, [Wiesler & Richard$^+$ 14] proposed a mean normalized stochastic gradient descent, and [Ioffe & Szegedy 15] proposed normalization of each layer's input during training. The process of normalizing each layer's input is called batch normalization. Given a batch $B$ with $m$ data vectors $x_i$, the process of batch normalization starts by calculating the mean $\mu_B$ and variance $\sigma_B^2$ of the batch B. Then, each input $x_i \in B$ is linearly transformed to have zero mean and unit variance as

$$\begin{aligned}
\mu_B &= \frac{1}{m} \sum_{i=1}^{m} x_i \ , \\
\sigma_B^2 &= \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_B)^2 \ , \\
\tilde{x}_i &= \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \ ,
\end{aligned}$$

where $\epsilon$ is a small real number to avoid divide by zero. Normalizing the output of a layer in this way, can limit the representational power of the layer. Hence, to cope with this problem, the normalized vectors $\tilde{x}_i$ are scaled and shifted to produce the output $y_i$ of batch normalization as

$$y_i = \gamma \tilde{x}_i + \beta \ ,$$

where $\gamma$ and $\beta$ are trainable parameters. Batch normalization is implemented as a layer in deep neural networks. It can be thought of whitening of a layer's input. Whitening is the process of linearly transforming input vectors to have zero mean and unit variance. With batch normalization, convergence speed of deep networks improve significantly.

# 4 Spatio Temporal Residual Networks

In this chapter, we describe our approach to address action recognition. The approach is mainly based on the deep residual network [He & Zhang$^+$ 15], called ResNet. ResNet reformulates a layer as learning the spatial residual function with respect to layer's input. State of the art results were achieved in image recognition tasks by learning spatial residual functions. We extend the approach to learn temporal residual functions across the frames to do action recognition in videos. The basics of ResNet are explained in Section 4.1, and finally, spatio temporal residual networks are described in Section 4.2.

## 4.1 ResNet

Deep neural networks have shown tremendous performance in computer vision tasks. They naturally integrate low, mid and high level features in end to end learning. It is evident in [Simonyan & Zisserman 14b] and [Szegedy & Liu$^+$ 14], that the depth of the network is of significant importance in complex learning tasks. However, training deep networks is not that easy. Vanishing or exploding gradient is one source that hamper the convergence of deep networks. One other problem that hinders the convergence of deep networks is the degradation problem, i.e. with increasing network depth accuracy gets saturated and then degrades.

Normalized initialization [Glorot & Bengio 10] can solve vanishing or exploding gradient. However, the degradation problem is hard to solve. It is evident in [He & Zhang$^+$ 15], that stacking more layers is not easy to optimize. They did an experiment, by picking a shallow network, and stacked a few more layers to it, by construction it should be possible that the deep network should not have more training error than the shallow network, considering the stacked layers are just doing the identity mapping. But their experiment showed that current solvers were unable tune the stacked layers to do identity mapping, and hence the training error was worse than in the shallow network.

The degradation problem is addressed by introducing a residual learning framework [He & Zhang$^+$ 15]. In residual learning framework few staked layers fit a residual mapping instead of the desired mapping. Let $H(x)$ be the desired mapping, with residual learning framework few stacked layers will learn residual mapping $F(x)$ instead of $H(x)$, and original mapping $H(x) = F(x) + x$. Figure 4.1 shows a building block of the residual learning framework.

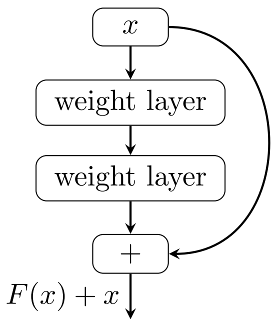


Figure 4.1: Residual learning building block. The residual mapping $F(x)$ is learned by stacked weight layers

Using the residual learning framework, deep networks of a few hundred layers can be optimized by stochastic gradient descent. In fact, the imagenet challenge 2015 [Russakovsky & Deng$^+$ 14] is won by a deep residual network of 152 layers [He & Zhang$^+$ 15].

## 4.2 Spatio Temporal Residual Network

Action recognition in videos is basically a task of video analysis. For action recognition, contextual information is needed. The contextual information in this case would be the information present in individual frames.

In practice, the feature vector for each individual frame is learned by any ConvNet architecture and then combined by some recurrent layer. As discussed earlier, if the video sequence is long, it is difficult to train a recurrent network.

Following the approach of ResNet, we propose, the feature vector of a frame at time step $t$ to be a residual function with respect to the frame at time step $t-1$. Figure 4.2 shows the building block of our spatio temporal residual learning framework. It is shown in Figure 4.2, how spatial and temporal residuals are combined.

The temporal context comes from the recurrent connection. With only one re-

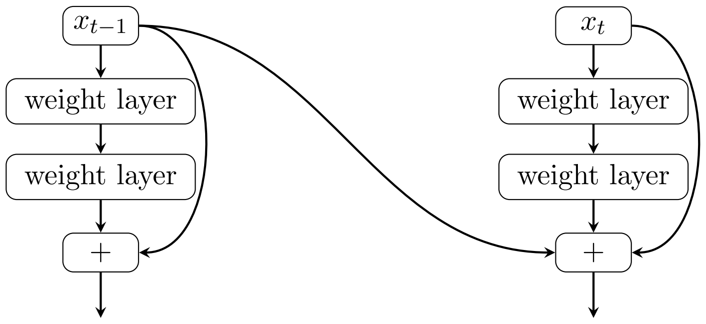


Figure 4.2: Building block of our spatio temporal residual learning, $x_{t-1}$ represents the input at time step $t-1$ and $x_t$ represents the input at time step $t$

current connection in the network, we exploit the dependency on two time steps. By following the analogy of ResNet to learn the temporal residuals, we need to create temporal skip connections in the ResNet. Figure 4.3 shows a spatio temporal residual network. The network has only one recurrent connection. Hence, it can exploit temporal dependencies over two consecutive frames. In an action recognition setting, the spatio temporal residual network maps sequence of frames to a single action class. Hence, the network output at the last time step is considered as the output for the complete sequence.

For such tasks, where the output depends on longer sequences, more recurrent connections are needed to have more temporal dependencies. Figure 4.4 shows a spatio temporal network, having two recurrent connections, that depict dependencies over three time steps. Similarly, the longer dependencies can be depicted by introducing more recurrent connections.

In order to use this approach for action recognition, a video is divided into $m$ small sequences each containing $T$ frames. A spatio residual network with $T-1$ recurrent connections is created, to capture the dependencies over $T$ time steps. For each small sequence $\{x_t^{(i)}\}_{t=1}^T$ within one video, the spatio temporal residual network computes $P(y = c|\{x_t^{(i)}\}_{t=1}^T)$. Individual output probabilities for each small sequence, computed by spatio temporal residual network, are averaged over $m$ sequences. Finally, the averaged probability distribution gives the probability for each action, happening in the video. Figure 4.5 shows the overall procedure of action recognition through the spatio temporal residual network.

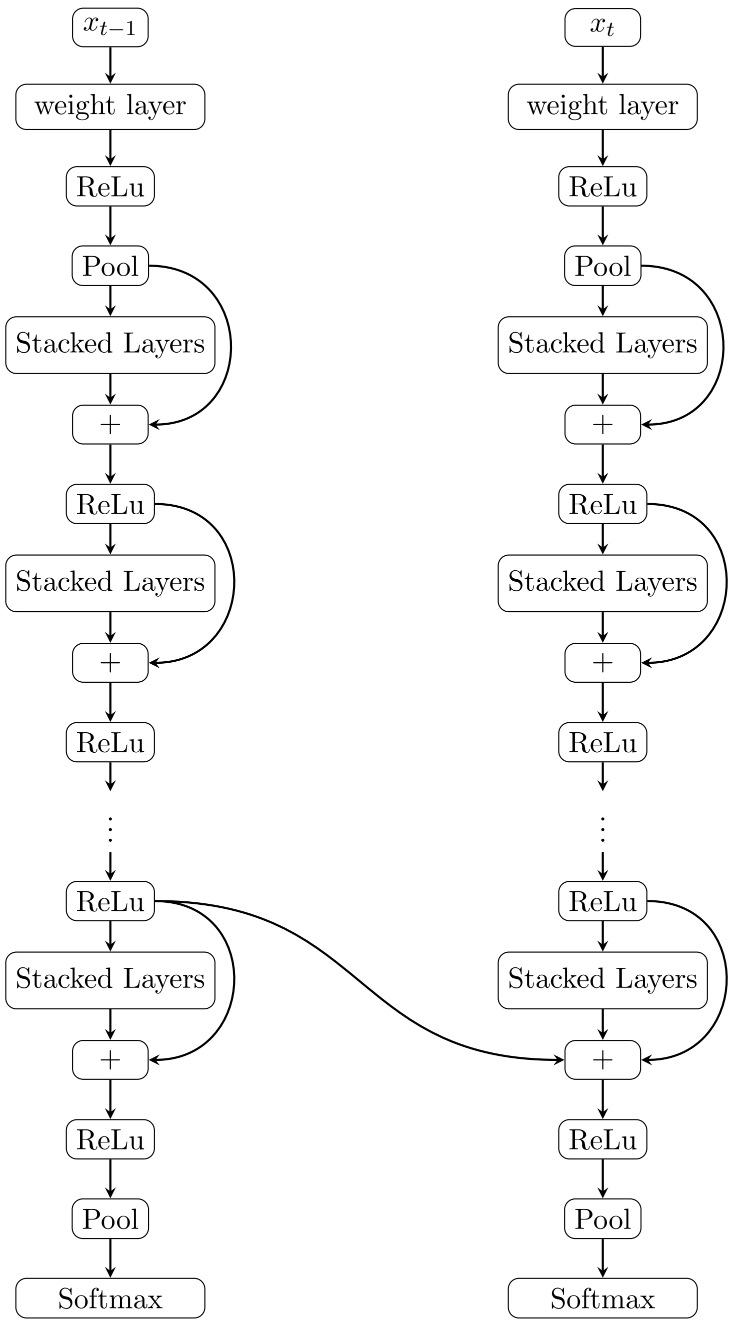


Figure 4.3: Spatio temporal network with one recurrent connection. The network is unfolded through time. It is capable of capturing temporal context between two consecutive frames.

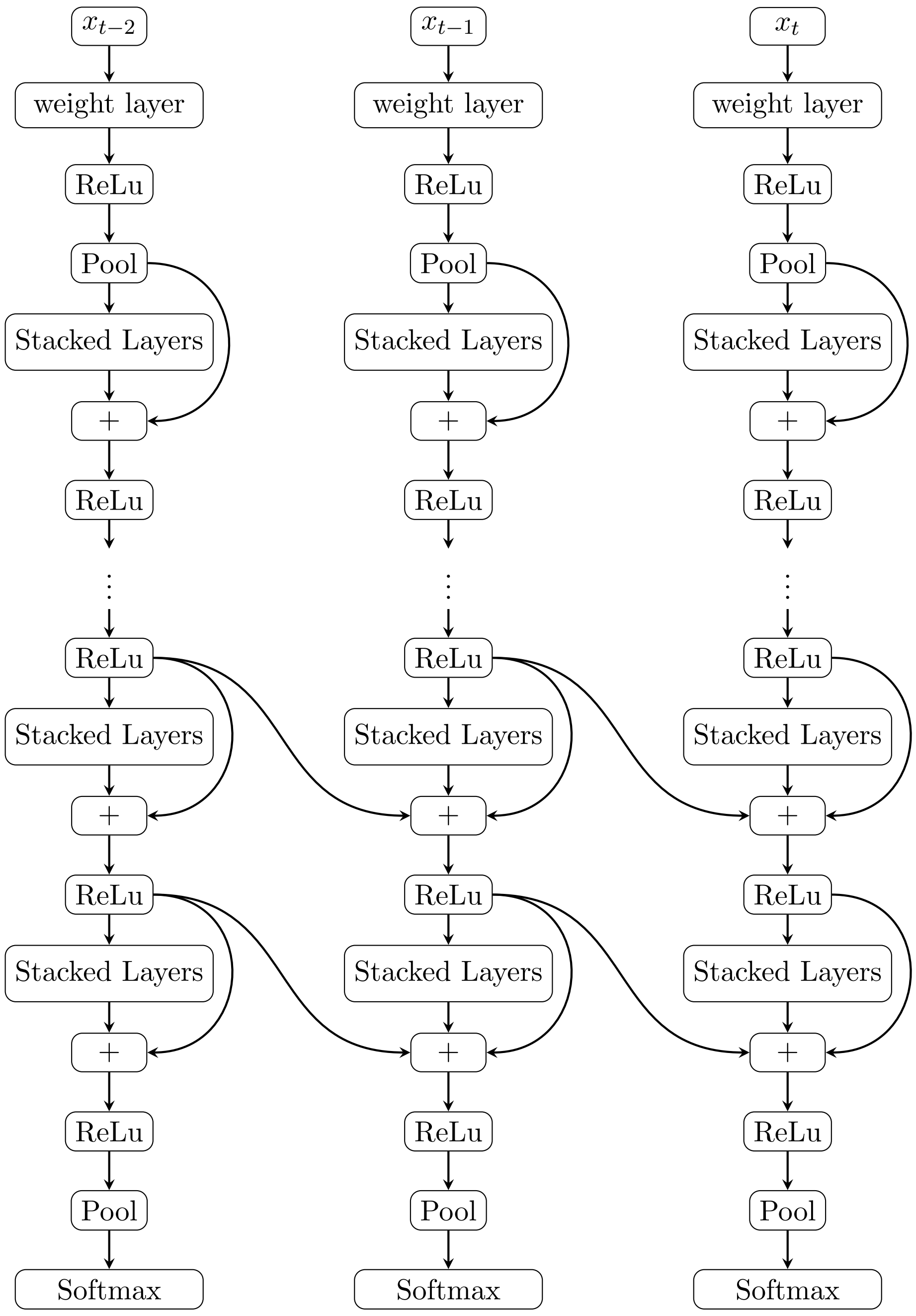


Figure 4.4: Spatio temporal network with two recurrent connections. The network is capable of capturing temporal context between three consecutive frames.

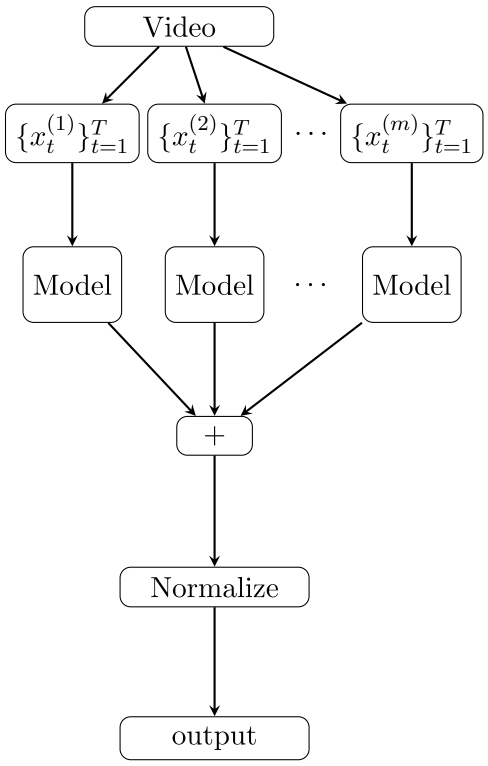


Figure 4.5: The overall procedure of classifying a video. Video is divided into $m$ smaller sequences each with $T$ frames. The output for each small sequence is calculated by spatio temporal network. Finally individual outputs for each small sequence are averaged to find the final probability distribution.

# 5 Experiments

This chapter the experiments are illustrated. We divide our experiments in four sets, i.e. to decide the best learning algorithm along with different hyper parameters, to decide the best position and type of recurrent skip connections, explore different sampling rates, and finally, the effect of the temporal context. In Section 5.1 we explain the dataset used for our experiments, in Section 5.2 we explain our baseline experiments, in Section 5.3 we investigate different learning algorithms along with different hyper parameters, in Section 5.4 we explore the position and type of recurrent skip connections, in Section 5.5 the effect of different sampling rates is investigated, and finally, in Section 5.6, the effect of an increase in temporal context is examined.

Except for the baseline experiments, unless if it is explicitly stated, we used ADAM as learning algorithm. Parameters are updated each time after processing 1% of data, and we sampled every tenth frame from each video.

## 5.1 UCF-101

UCF-101 [Soomro & Zamir$^{+}$ 12] is a large scale action recognition dataset. The dataset contains realistic videos collected from YouTube. With 13,320 videos from 101 action classes, UCF-101 gives the largest diversity in terms of actions, and with the presence of large variations in camera motion, object appearance and pose, object scale, viewpoint, cluttered background, illumination conditions etc, it is the most relevant dataset to date.

Videos with 101 classes are further grouped into 25 groups. Each group contains four to seven videos. The videos from the same group share some common features, such as similar background, and similar viewpoint etc.

## 5.2 Baseline Experiments

In this section we explain our baseline experiments. As a baseline, we extracted imagenet features for individual frames in the video. Averaged individual feature

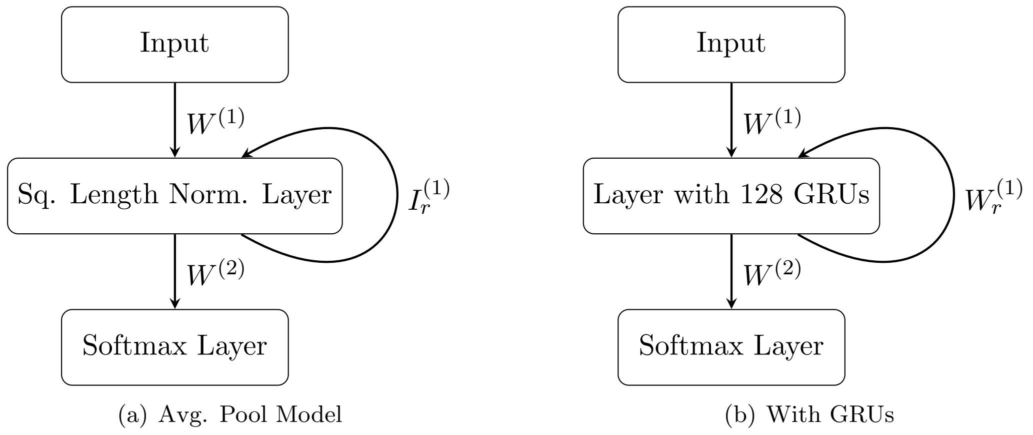


Figure 5.1: Architectures used for baseline experiments. In (a) $I_r^{(1)}$ represents untrainable identity mapping weights, and Sq. Length Norm. Layer is averaging feature vectors over complete sequence. In (b) GRUs represents gated recurrent units [Cho & van Merrienboer+ 14]

vectors represent the feature vector of the complete video. Feature vectors for individual frames are extracted using a pre-trained ResNet (see Section 4.1) model with 50 layers. A batch normalization layer is added after each layer, to normalize the input to a layer. This way the network has in total 106 layers.

For baseline experiments, we extracted the imagenet feature vectors for each frame at three different positions of ResNet, i.e. after the 105th layer, we call these features block4 features, after the 85th layer, called block3 features, and after the 48th layer, called block2 features. The layers considered for feature extraction, are the positions in the network where spatial down sampling happens.

We performed two sets of experiments on extracted features for each block. In one set, we average frame level feature vectors, after Z-Normalization and without Z-Normalization, and train a linear classifier. Similarly, in the other set we use a recurrent neural network with 128 gated recurrent units. Figure 5.1 shows two architectures used for the baseline experiments. We use ADAM (see Section 3.2) as learning algorithm with 0.01 as learning rate. Table 5.1, shows the baseline experiments on imagenet features. It shows that the average pooling model outperforms the model with gated recurrent units. Also, it is evident from the experiments, with more depth, features become richer. Hence, the depth plays significant role in getting good classification accuracy.

| Features | Method | Error Rate (with Z-Norm) | Error Rate (without Z-Norm) |
|---|---|---:|---:|
| Block4 | Avg. Pool | **0.236** | 0.237 |
| Block4 | GRU | 0.239 | 0.276 |
| Block3 | Avg. Pool | 0.309 | 0.313 |
| Block3 | GRU | 0.403 | 0.325 |
| Block2 | Avg. Pool | 0.431 | 0.434 |
| Block2 | GRU | 0.440 | 0.493 |

Table 5.1: Results of the baseline experiments

As another baseline, we finetuned a ResNet pre-trained on imagenet, on individual frames of videos from UCF-101. As on imagenet features, the average pooling model performed best among both models. Hence, in these experiments we only considered the average pooling model. Also, we only considered the features from block4 (after the 105th layer), and block3 (after the 85th layer). Table 5.2 shows the baseline experiments, with features extracted from finetuned model, and provide a comparison with the results achieved with imagenet features.

| Features | Error Rate (with Z-Norm) | Error Rate (without Z-Norm) |
|---|---:|---:|
| Block4 | **0.236** | 0.237 |
| Block4 (Finetuned) | 0.260 | 0.255 |
| Block3 | 0.309 | 0.313 |
| Block3 (Finetuned) | 0.389 | 0.410 |

Table 5.2: Results of the baseline experiments, on finetuned model

The results achieved by the features from the finetuned model are significantly worse than the imagenet features. This is due to low inter class variation of individual frames. During finetuning, it may happen that some frames are falsely classified as some other class than the class of video it belongs to. Hence, due to sensitivity of average pooling to outliers, the video is misclassified.

## 5.3 Investigation of learning algorithms

In this set of experiments, we explore the best learning algorithm to be used for spatio temporal residual networks. Because of the significant depth of ResNet, memory limitation is one of the challenges, to be faced while training spatio temporal residual network, while it is required to set a sufficiently large batch size.

As discussed earlier, for stochastic gradient descent, the loss fluctuates a lot during the course of training. This is mainly due the fact that to parameter updates are based on stochastic approximations rather than on the actual gradient. With a smaller batch size, the gradient approximations tend to be more noisy. So, it might happen that learning does not converge or convergence is achieved very slowly. Hence, good convergence is achieved with a large enough batch size.

Another problem related to a small batch size, particular to video analysis, is that most likely the data within one batch belongs to a single video or not more than two or three videos due to GPU memory limitations. This causes even more noise in the gradient approximation as before approximating the gradient, the model has not seen all or enough classes. Due to these issues convergence slows down.

So, we process the input in small batches, as that is due to limited memory. But we update the parameters after considerable percentage of the data is processed by the model. In these experiments, we investigate the best learning algorithm among RPROP or ADAM (see Section 3.2), also we tried to find out after how much data parameters should be updated. As RPROP is a batch learning method we only consider updating after 10% of data, but for ADAM we consider two update strategies, i.e. after processing 10% of data and after 1% of data.

Figure 4.3 shows the network architecture used for these experiments. The network has only one recurrent connection, hence it captures the dependency over two frames only (as discussed in Section 4.2). The recurrent connection is from the 98th layer to the 104th layer, after the 104th layer, there is only an average pooling layer and softmax output layer. Also, the recurrent connection has a non trainable identity mapping as weights.

| Algorithm | Update After | Error Rate |
|---|---:|---:|
| ADAM | 1% | **0.197** |
| ADAM | 10% | 0.233 |
| RPROP | 10% | 0.222 |

Table 5.3: Results achieved by different learning algorithms, along with different update strategies

Table 5.3, shows the results of experiments performed in order to select better learning algorithm, along with a better update strategy. It is evident from the results, that ADAM performed best, with an update strategy of updating parameters after processing 1% of the data.

## 5.4 Effect of type and position of the recurrent connection

In this set of experiments, we evaluate different types of recurrent connections, along with their position in the network. We evaluate recurrent connections, at four different positions, i.e. at the beginning after few layers in the network (from the 5th to the 11th layer), in the middle of the network (from the 49th to the 55th layer), and in the end of the network (from the 86th to the 92nd layer and from the 98th to the 104th layer). Also we evaluated the effect of type of recurrent connections. Up until now we only evaluated identity mapping recurrent connections, in these experiments, we evaluate recurrent connections, with convolutional weights and kernels of size $1 \times 1$.

| Position | Type | Error Rate |
|---|---|---|
| 5-11 | Convolutional | 0.265 |
| 49-55 | Convolutional | 0.234 |
| 86-92 | Convolutional | 0.231 |
| 98-104 | Identity Mapping | 0.197 |

Table 5.4: Results for placing the recurrent connection at different positions in the network

Table 5.4 shows, the deeper we place the recurrent connection in the network, the better is the classification accuracy. However, none of the configurations (with convolutional weights) in the Table 5.4, achieved better performance than an identity mapping recurrent connection from the 98th to the 104th layer, considered in previous experiments.

Hence, we update the configuration of the connection from the 98th to 104th layer. In the new configuration, the connection performs a parametrized linear or non linear transformation, rather than an identity mapping. We consider two new configurations, i.e. the connection is doing linear transformation (convolution with kernel of size $1 \times 1$), and convolution followed by non-linear transformation.

| Type | Error Rate |
|---|---|
| Identity Mapping | **0.197** |
| Conv. Linear | 0.219 |
| Conv. Non-Linear | 0.210 |

Table 5.5: Results achieved by different type of recurrent connections

Table 5.5 shows the results achieved by different type of connections, all placed closer to the output layer as our previous analysis shows that works best. Identity mapping connection with non trainable weights performed best, possibly because introducing more weights in the network causes overfitting.

## 5.5 Effect of Sampling Rate

Until now, in all the experiments, we considered every tenth frame of the video as an input. In this set of experiments we evaluate the effect of the sampling rate. We consider four different sampling rates, i.e. sampling every second, every fifth, every tenth and every fifteenth frame, respectively. We applied the best architecture so far, i.e. one identity mapping recurrent connection, between the 98th and 104th layer, to all discussed sampling techniques.

| Sampling Rate | Error Rate | Approx. Training Time |
|---|---:|---:|
| 02 | **0.195** | 161 Hrs |
| 05 | **0.196** | 61 Hrs |
| 10 | **0.197** | 35 Hrs |
| 15 | 0.213 | 23 Hrs |

Table 5.6: Results achieved by different sampling rates.

Table 5.6 shows the results. Although the best results are achieved by sampling every second frame in the video, the difference in error rates between sampling every second, every fifth, and every tenth is almost negligible. This is mainly because, consecutive frames in the videos, are usually visually very similar. With increase in sampling there is not much gain in terms accuracy, however it adds significant amount of time in training. So, this set of experiments gives evidence the best sampling rate considering training time and accuracy, is every tenth frame of the video.

## 5.6 Effect of Temporal Context

In this set of experiments, we explore the effect of temporal context. As discussed earlier with more recurrent connections, the network is able to include additional temporal dependencies. We already investigated, network with one recurrent connection, that is able to include temporal context of two frames. In these experiments, we further explore, the temporal context of three frames (by two recurrent connections in the network), and the temporal context of five frames (by four recurrent connections in the network).

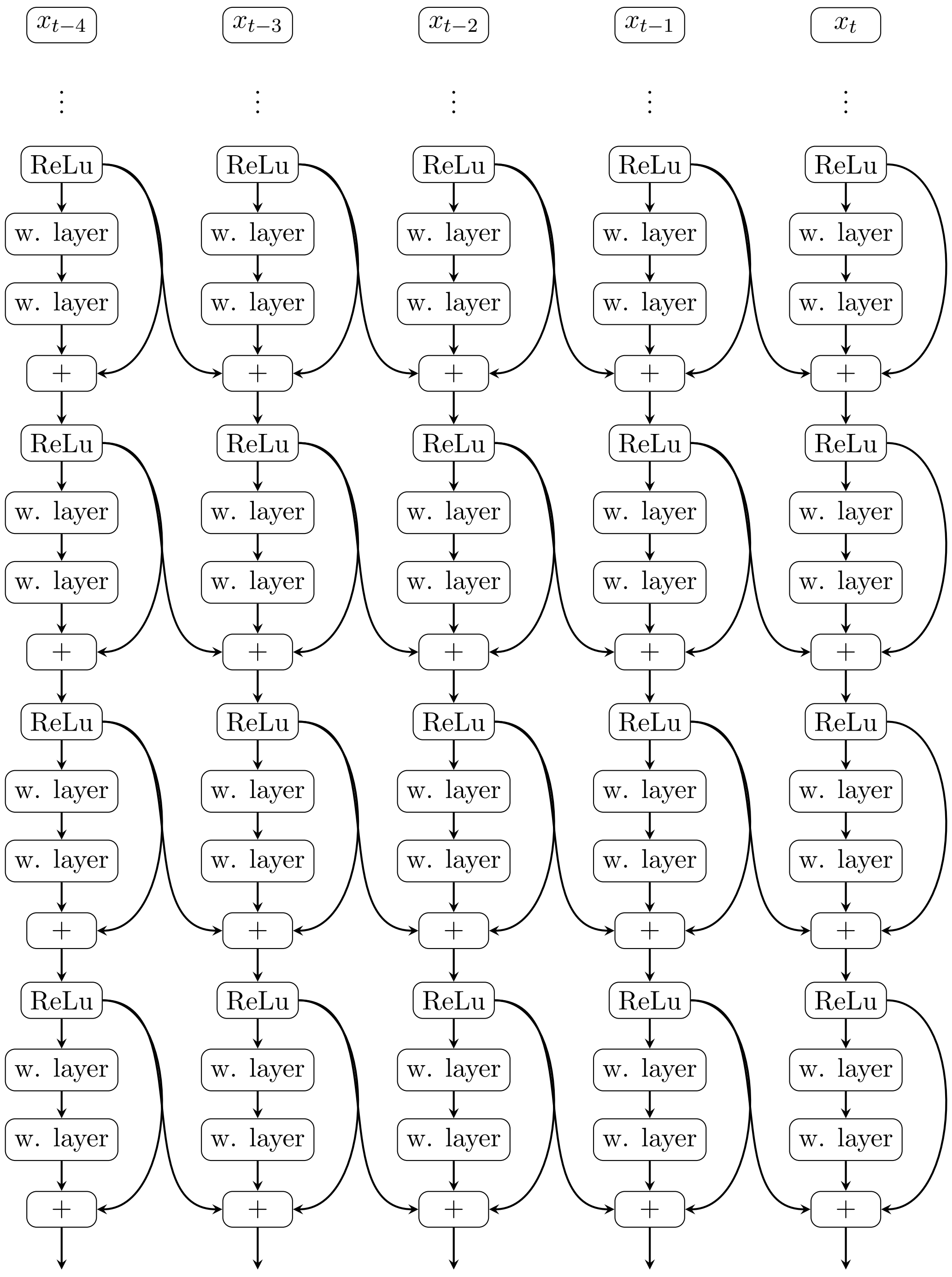


Figure 5.2: Network architecture to include temporal context of five frames

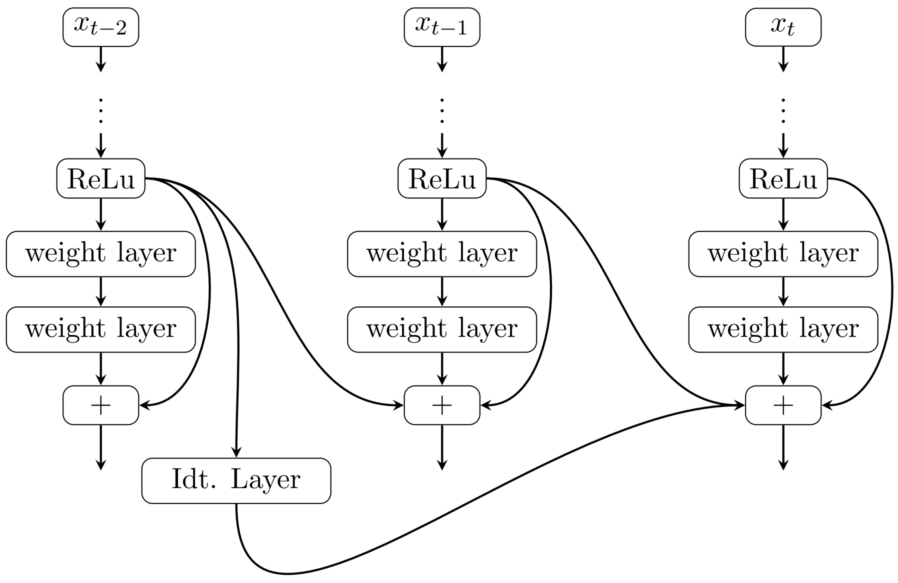


Figure 5.3: Slightly different way to include more temporal context. The connection from the Idt. Layer (Identity Layer) skips the frame at time $x_{t-1}$

Figures 4.4 and 5.2 (due to lack of space, some layers are omitted) show network architectures, to accommodate temporal context over three and five frames respectively. In these experiments we also explored a slightly different model to include temporal dependencies over three frames, we call this model Special3, shown in Figure 5.3. The Special3 model is similar to the best model so far (with one recurrent connection from the 98th to 104th layer). However, in Special3 model we introduce an identity layer and connect the 98th layer in the network to the identity layer through a recurrent connection, and the identity layer is connected to the 104th layer in the network through an other recurrent connection, this also enables the model to capture temporal dependencies three frames.

| Temporal Context | Model | Error Rate |
|---|---|---|
| 2 | 1 recurrent connection | **0.197** |
| 3 | 2 recurrent connections | **0.194** |
| 3 | Special3 | 0.205 |
| 5 | 4 recurrent connections | 0.209 |

Table 5.7: Results achieved by including more temporal context

As it is evident in Table 5.7, we do not gain much by including more temporal context. The accuracy improves in case of temporal context three, however it gets worse in case of temporal context five. Hence, considering training time, we consider the model with only one recurrent connection as the best model.

We further evaluate our best model on all three splits of UCF-101.

| Split | Error Rate |
|---|---|
| 1 | 0.197 |
| 2 | 0.205 |
| 3 | **0.192** |

Table 5.8: Results achieved on all three splits of UCF-101

As Table 5.8 summarizes, on average our best model achieves **0.198** on UCF-101 [Soomro & Zamir$^+$ 12].

## 5.7 Comparison with the State of the art

In this section, our best model (with one temporal skip connection and with sample rate 10) is compared with state of the art action recognition methods. As motion in the frames and appearance in individual frames are two complementary aspects for action recognition. Hence, most of the state of the art methods consider two different neural networks (appearance stream and motion stream) to extract and use appearance and motion for action recognition. The output of both the networks is fused, and a simple classifier is trained to classify videos. As our model fuse appearance information across temporal axis, so mainly our comparison is with the appearance streams in the state of the art models, but for completeness we will also compare our results with results achieved after the outputs of the appearance and the motion streams are fused.

| Method | Appearance Stream | Motion Stream | Final |
|---|---|---|---|
| Improved Dense Trajectories [Wang & Schmid 13] | - | - | 0.141 |
| Two Stream Architecture [Simonyan & Zisserman 14a] | 0.270 | 0.163 | 0.120 |
| Two Stream Architecture (GoogleNet) [Wang & Xiong$^+$ 15] | 0.247 | 0.142 | 0.107 |
| Two Stream Architecture (VGG-Net) [Wang & Xiong$^+$ 15] | 0.216 | **0.130** | **0.086** |
| Spatiotemporal ResNets [Feichtenhofer & Pinz$^+$ 16] | **0.177** | 0.209 | 0.105 |
| **Spatio Temporal Residual Networks** | 0.198 | - | 0.198 |

Table 5.9: Comparison with the state of the art

Table 5.9 summarizes the comparison with the state of the art. Even appearance stream of [Feichtenhofer & Pinz$^+$ 16] outperform all other approaches, it is not purely appearance stream as there is a connection from motion stream to appearance stream in [Feichtenhofer & Pinz$^+$ 16]. Hence, our model outperforms appearance stream of all other state of the art models.

# 6 Conclusion

This chapter conclude our work. To summarize our work, we first tried to find better learning algorithm for our spatio temporal residual networks, along with better hyper parameters i.e. when to update the model. We found out that ADAM is better learning algorithm and update to the model should be done after processing 1% of data. We tried to determine the optimal position and type (convolutional connection, or identity mapping connection) of the temporal skip connection (the recurrent connections in spatio temporal residual networks), and we found the later we put temporal skip connection the better the spatio temporal features will be, and to avoid overfitting identity mapping connection should be used as temporal skip connection. We explored the effect of sampling rate, and we found densely sampling every other frame will not help as there isn't much visual variation in the frames, which are closer to each other across temporal axis. Effect of temporal context is also explored and we observed that there isn't much gain from increasing temporal context from 2 to 3 however, the results get worse for temporal context 5.

Information about motion across the frames is really important to achieve the state of the art results for action recognition. Our model can process the motion input in the form of a stack of optical flow field. As our model outperforms state of the art appearance streams, as a future work it is worth trying two different spatio temporal residual networks, for motion and appearance streams respectively. We expect similar gain in terms accuracy for motion stream and after fusion of both appearance and motion streams.

# 7 List of Figures

# 8 List of Tables

# Bibliography

[Bengio & Boulanger-Lewandowski$^+$ 12] Y. Bengio, N. Boulanger-Lewandowski, R. Pascanu. Advances in optimizing recurrent networks. *CoRR*, Vol. abs/1212.0901, 2012.

[Bengio & Simard$^+$ 94] Y. Bengio, P. Simard, P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *Trans. Neur. Netw.*, Vol. 5, No. 2, pp. 157–166, March 1994.

[Bilen & Fernando$^+$ 16] H. Bilen, B. Fernando, E. Gavves, A. Vedaldi, S. Gould. Dynamic image networks for action recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[Chen & Ting$^+$ 10] B. Chen, J.-A. Ting, B. Marlin, N. de Freitas. Deep learning of invariant spatio-temporal features from video. In *NIPS 2010 Deep Learning and Unsupervised Feature Learning Workshop*, 2010.

[Cho & van Merrienboer$^+$ 14] K. Cho, B. van Merrienboer, D. Bahdanau, Y. Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *CoRR*, Vol. abs/1409.1259, 2014.

[deeplearning.net] deeplearning.net. Feature maps produced by convolutional layer.

[Donahue & Hendricks$^+$ 14] J. Donahue, L. A. Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, T. Darrell. Long-term recurrent convolutional networks for visual recognition and description. *CoRR*, Vol. abs/1411.4389, 2014.

[Duchi & Hazan$^+$ 11] J. Duchi, E. Hazan, Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, Vol. 12, pp. 2121–2159, July 2011.

[Feichtenhofer & Pinz$^+$ 16] C. Feichtenhofer, A. Pinz, R. P. Wildes. Spatiotemporal residual networks for video action recognition. *CoRR*, Vol. abs/1611.02155, 2016.

[Fernando & Gavves$^+$ 15] B. Fernando, E. Gavves, J. O. M., A. Ghodrati, T. Tuytelaars. Rank pooling for action recognition. *CoRR*, Vol. abs/1512.01848, 2015.

[Glorot & Bengio 10] X. Glorot, Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *JMLR W&CP: Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS 2010)*, Vol. 9, pp. 249–256, May 2010.

[Glorot & Bordes$^+$ 11] X. Glorot, A. Bordes, Y. Bengio. Deep sparse rectifier neural networks. In G. J. Gordon, D. B. Dunson, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS-11)*, Vol. 15, pp. 315–323. Journal of Machine Learning Research -Workshop and Conference Proceedings, 2011.

[He & Zhang$^+$ 15] K. He, X. Zhang, S. Ren, J. Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, Vol., 2015.

[Hochreiter & Schmidhuber 97] S. Hochreiter, J. Schmidhuber. Long short-term memory. *Neural Comput.*, Vol. 9, No. 8, pp. 1735–1780, Nov. 1997.

[Hornik & Stinchcombe$^+$ 89] K. Hornik, M. Stinchcombe, H. White. Multilayer feedforward networks are universal approximators. *Neural Netw.*, Vol. 2, No. 5, pp. 359–366, July 1989.

[Hubel & Wiesel 68] D. H. Hubel, T. N. Wiesel. Receptive fields and functional architecture of monkey striate cortex. *Journal of Physiology (London)*, Vol. 195, pp. 215–243, 1968.

[Idrees & Zamir$^+$ 16] H. Idrees, A. R. Zamir, Y. Jiang, A. Gorban, I. Laptev, R. Sukthankar, M. Shah. The THUMOS challenge on action recognition for videos "in the wild". *CoRR*, Vol. abs/1604.06182, 2016.

[Ioffe & Szegedy 15] S. Ioffe, C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, Vol. abs/1502.03167, 2015.

[Jain & van Gemert$^+$ 15] M. Jain, J. C. van Gemert, C. G. M. Snoek. What do 15,000 object categories tell us about classifying and localizing actions? In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[Jhuang & Serre$^+$ 07] H. Jhuang, T. Serre, L. Wolf, T. Poggio. A biologically inspired system for action recognition. In *International Conference on Computer Vision (ICCV)*, 2007.

[Ji & Xu$^+$ 13] S. Ji, W. Xu, M. Yang, K. Yu. 3d convolutional neural networks for human action recognition. *IEEE Trans. Pattern Anal. Mach. Intell.*, Vol. 35, No. 1, pp. 221–231, Jan. 2013.

[Karpathy & Toderici$^+$ 14] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, L. Fei-Fei. Large-scale video classification with convolutional neural networks. In *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '14, pp. 1725–1732, Washington, DC, USA, 2014. IEEE Computer Society.

[Ken Chatfield & Zisserman 11] A. V. Ken Chatfield, Victor Lempitsky, A. Zisserman. The devil is in the details: an evaluation of recent feature encoding methods. In *Proceedings of the British Machine Vision Conference*, pp. 76.1–76.12. BMVA Press, 2011. http://dx.doi.org/10.5244/C.25.76.

[Kingma & Ba 14] D. P. Kingma, J. Ba. Adam: A method for stochastic optimization. *CoRR*, Vol. abs/1412.6980, 2014.

[Laptev & Marszalek$^+$ 08] I. Laptev, M. Marszalek, C. Schmid, B. Rozenfeld. Learning realistic human actions from movies. In *CVPR*. IEEE Computer Society, 2008.

[Le & Zou$^+$ 11] Q. V. Le, W. Y. Zou, S. Y. Yeung, A. Y. Ng. Learning hierarchical invariant spatio-temporal features for action recognition with independent subspace analysis. In *CVPR*, pp. 3361–3368. IEEE Computer Society, 2011.

[LeCun & Haffner$^+$ 99] Y. LeCun, P. Haffner, L. Bottou, Y. Bengio. Object recognition with gradient-based learning. In D. Forsyth, editor, *Feature Grouping*. Springer, 1999.

[Oneata & Verbeek$^+$ 13] D. Oneata, J. Verbeek, C. Schmid. Action and event recognition with fisher vectors on a compact feature set. In *Proceedings of the 2013 IEEE International Conference on Computer Vision*, ICCV '13, pp. 1817–1824, Washington, DC, USA, 2013. IEEE Computer Society.

[Peng & Zou$^+$ 14] X. Peng, C. Zou, Y. Qiao, Q. Peng. *Action Recognition with Stacked Fisher Vectors*, pp. 581–595. Springer International Publishing, Cham, 2014.

[Perronnin & Sánchez$^+$ 10] F. Perronnin, J. Sánchez, T. Mensink. Improving the fisher kernel for large-scale image classification. In *Proceedings of the 11th European Conference on Computer Vision: Part IV*, ECCV'10, pp. 143–156, Berlin, Heidelberg, 2010. Springer-Verlag.

[Richard & Gall 16] A. Richard, J. Gall. Temporal action detection using a statistical language model. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[Riedmiller & Braun 93] M. Riedmiller, H. Braun. A direct adaptive method for faster backpropagation learning: The rprop algorithm. In *IEEE INTERNATIONAL CONFERENCE ON NEURAL NETWORKS*, pp. 586–591, 1993.

[Riesenhuber & Poggio 99] M. Riesenhuber, T. Poggio. Hierarchical models of object recognition in cortex. *Nature neuroscience*, Vol. 2, No. 11, pp. 1019–1025, 1999.

[Rohrbach & Amin$^+$ 12] M. Rohrbach, S. Amin, M. Andriluka, B. Schiele. A database for fine grained activity detection of cooking activities. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, IEEE, June 2012.

[Rumelhart & Hinton$^+$ 88] D. E. Rumelhart, G. E. Hinton, R. J. Williams. Neurocomputing: Foundations of research. chapter Learning Internal Representations by Error Propagation, pp. 673–695. MIT Press, Cambridge, MA, USA, 1988.

[Russakovsky & Deng$^+$ 14] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. S. Bernstein, A. C. Berg, F. Li. Imagenet large scale visual recognition challenge. *CoRR*, Vol. abs/1409.0575, 2014.

[Shimodaira 00] H. Shimodaira. Improving predictive inference under covariate shift by weighting the log-likelihood function. *Journal of Statistical Planning and Inference*, Vol. 90, No. 2, pp. 227–244, Oct. 2000.

[Siegelmann & Sontag 92] H. T. Siegelmann, E. D. Sontag. On the computational power of neural nets. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, COLT '92, pp. 440–449, New York, NY, USA, 1992. ACM.

[Simonyan & Zisserman 14a] K. Simonyan, A. Zisserman. Two-stream convolutional networks for action recognition in videos. *CoRR*, Vol. abs/1406.2199, 2014.

[Simonyan & Zisserman 14b] K. Simonyan, A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, Vol. abs/1409.1556, 2014.

[Soomro & Zamir$^+$ 12] K. Soomro, A. R. Zamir, M. Shah. UCF101: A dataset of 101 human actions classes from videos in the wild. *CoRR*, Vol. abs/1212.0402, 2012.

[Szegedy & Liu$^+$ 14] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich. Going deeper with convolutions. *CoRR*, Vol. abs/1409.4842, 2014.

[Taylor & Fergus$^+$ 10] G. W. Taylor, R. Fergus, Y. LeCun, C. Bregler. Convolutional learning of spatio-temporal features. In *Proceedings of the 11th European Conference on Computer Vision: Part VI*, ECCV'10, pp. 140–153, Berlin, Heidelberg, 2010. Springer-Verlag.

[Tieleman & Hinton 12] T. Tieleman, G. Hinton. Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.

[Wang & Klaser$^+$ 11] H. Wang, A. Klaser, C. Schmid, C.-L. Liu. Action recognition by dense trajectories. In *Proceedings of the 2011 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '11, pp. 3169–3176, Washington, DC, USA, 2011. IEEE Computer Society.

[Wang & Qiao$^+$ 15] L. Wang, Y. Qiao, X. Tang. Action recognition with trajectory-pooled deep-convolutional descriptors. *CoRR*, Vol. abs/1505.04868, 2015.

[Wang & Schmid 13] H. Wang, C. Schmid. Action Recognition with Improved Trajectories. In *ICCV 2013 - IEEE International Conference on Computer Vision*, pp. 3551–3558, Sydney, Australia, Dec. 2013. IEEE.

[Wang & Xiong$^+$ 15] L. Wang, Y. Xiong, Z. Wang, Y. Qiao. Towards good practices for very deep two-stream convnets. *CoRR*, Vol. abs/1507.02159, 2015.

[Wang & Xiong$^+$ 16] L. Wang, Y. Xiong, Z. Wang, Y. Qiao, D. Lin, X. Tang, L. V. Gool. Temporal segment networks: Towards good practices for deep action recognition. *CoRR*, Vol. abs/1608.00859, 2016.

[Weinzaepfel & Martin$^+$ 16] P. Weinzaepfel, X. Martin, C. Schmid. Towards weakly-supervised action localization. *CoRR*, Vol. abs/1605.05197, 2016.

[Wiesler & Richard$^+$ 14] S. Wiesler, A. Richard, R. Schl
"uter, H. Ney. Mean-normalized stochastic gradient for large-scale deep learning. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, pp. 180–184, Florence, Italy, May 2014. IBM Research Spoken Language Processing Student Travel Grant Award.

[Yang & Molchanov$^+$ 16] X. Yang, P. Molchanov, J. Kautz. Multilayer and multimodal fusion of deep neural networks for video classification. In *Proceedings of the 2016 ACM on Multimedia Conference*, MM '16, pp. 978–987, New York, NY, USA, 2016. ACM.

[Yeung & Russakovsky$^+$ 15] S. Yeung, O. Russakovsky, G. Mori, L. Fei-Fei. End-to-end learning of action detection from frame glimpses in videos. *CoRR*, Vol. abs/1511.06984, 2015.