(Draft : March 22, 2018)

Technische Universität Berlin

Institut für Softwaretechnik und Theoretische Informatik
Maschinelles Lernen / Intelligente Datenanalyse

Fakultät IV – Elektrotechnik und Informatik
Franklinstrasse 28-29
10587 Berlin
http://www.ml.tu-berlin.de

Master Thesis

Designing Recurrent Neural Networks for Explainability

Pattarawat Chormai

Matriculation Number: 387441
31.03.2018

Supervisor
Prof. Dr. Klaus-Robert Müller

Advisor
Dr. Grégoire Montavon

Acknowledgement

First of all, I would like to thank Prof. Dr. Klaus-Robert Müller and Dr. Grégoire Montavon for this research opportunity and invaluable guidance throughout the course of conducting the thesis as well as facilitating me at TU Berlin, Machine Learning group with a great research environment . I would also like to thank Prof. Dr. Klaus Obermayer and staffs of Neural Information Processing group for organizing Machine Intelligence I & II and Neural Information Project. These courses provide me necessary knowledge to conduct the thesis.

Secondly, I would like to thank to my family for always providing me financial and mental support.

Thanks EIT colleagues and . In particular, Someone for proofreading..

I would like to thank EIT Master School as well as TU/e and TUB staffs that involve in this study program. Your support and advice are always helpful.

Lastly, I would like to acknowledge Amazon AWS for generous educational credits and Auction Club S.a.r.L, Luxembourg for lending me a powerful laptop to use in my study. None of experiments would have been possible without these computational resource supports.

Hereby I declare that I wrote this thesis myself with the help of no more than the mentioned literature and auxiliary means.

Berlin, 15.03.2018

. . . . . .

Pattarawat Chormai

Abstract

Standard (non-LSTM) recurrent neural networks have been challenging to train, but special optimization techniques such as heavy momentum makes this possible. However, the potentially strong entangling of features that results from this difficult optimization problem can cause deep Taylor or LRP-type to perform rather poorly due to their lack of global scope. LSTM networks are an alternative, but their gating function make them hard to explain by deep Taylor LRP in a fully principled manner. Ideally, the RNN should be expressible as a deep ReLU network, but also be reasonably disentangled to let deep Taylor LRP perform reasonably. The goal of this thesis will be to enrich the structure of the RNN with more layers to better isolate the recurrent mechanism from the representational part of the model.

Zusammenfassung

Da die meisten Leuten an der TU deutsch als Muttersprache haben, empfiehlt es sich, das Abstract zusï¿½tzlich auch in deutsch zu schreiben. Man kann es auch nur auf deutsch schreiben und anschlieï¿½end einem Englisch-Muttersprachler zur ï¿½bersetzung geben.

Contents

Notation

$\boldsymbol{\theta}$	Parameters of a neural network
$\boldsymbol{x}, \boldsymbol{x}^{(\alpha)}$	A vector representing an input sample
σ	An activation function
$\{a_j\}_L$	A vector of activations of neurons in layer L
a_j	Activation of neuron j
b_k	Bias of neuron k
R_j	Relevance score of neuron j
$R_{j \leftarrow k}$	Relevance score distributed from neuron k to neuron j
w_{jk}	Weight between neuron j to neuron k
x_i	Feature i of input sample x

List of Figures

List of Tables

1 Introduction

2 Related Work

3 Background

3.1 Neural Networks

Neural networks(NNs) are a type of machine learning algorithms that inspired by how human brain works. In particular, NNs have units called neurons connecting together similar to the way of neurons our brain do. These connections allow NNs to hierarchically build representations that are necessary to perform an objective task. Figure 3.1 illustrates a simple coordination reaction task by our brain.

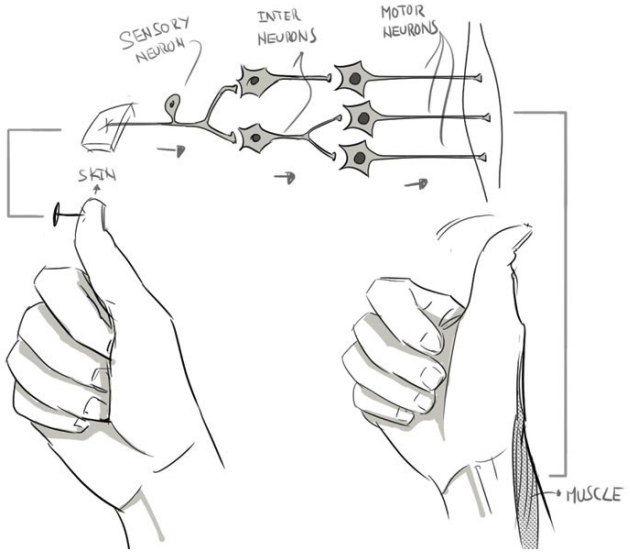


Figure 3.1: An illustration of how neurons in our brain cooperate together to sense the pain and move the thumb. [1]

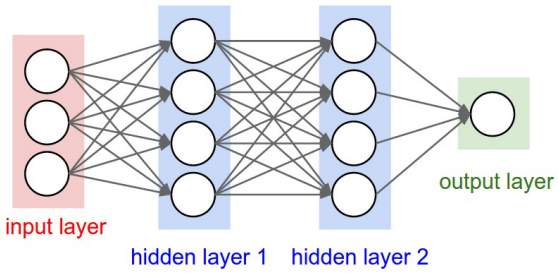


Figure 3.2

Figure 3.2 illustrates a general structure of NNs. It has input layer, output layer and hidden layers, which are analogously similar to sensory, motor and inter neurons in

[1] source: Eugenio N. Leon, `https://becominghuman.ai/making-a-simple-neural-network-2ea1de81ec20`

Figure 3.1. The goal is to build connections between these neurons such that the NN is able to perform an objective task with high accuracy. This process is called *training*. In this example, the objective task to classify what is the given digit.

Consider a given set of p training samples $\mathcal{D} = \{\boldsymbol{x}^{(\alpha)}, y^{(\alpha)})\}_{\alpha=1}^{p}$, there are 3 primary components to train a NN, namely

1. **Network architecture** : it defines how neurons connect and communicate to each other. More precisely, these are corresponding to connection weights and biases $\boldsymbol{\theta}$ and neuron's activation functions illustrated in Figure 3.2. Mathematically, it is a function f with parameters $\boldsymbol{\theta}$ that nonlinearly transforms an input $\boldsymbol{x}^{(\alpha)} \in \mathbb{R}^d$ to some values.

2. **Loss function** L : it is a measurement corresponding to the objective task that quantifies whether output $f(\boldsymbol{x}^{(\alpha)})$ from the NN matches the true output $y^{(\alpha)}$.

3. **Learning algorithm** : it is responsible to find the network's parameters $\hat{\boldsymbol{\theta}}$ to optimize the loss function L(*Empirical Error*) as a proxy for optimizing loss or error for unforeseen $\boldsymbol{x} \notin \mathcal{D}$ (*Generalization Error*).

Hence, the goal of this empirical training process can be summarized as follows :

$$\hat{\boldsymbol{\theta}} = \underset{\theta}{\operatorname{argmin}} \sum_{\alpha=1}^{p} L(f(\boldsymbol{x}^{(\alpha)}), y^{(\alpha)}) \tag{3.1}$$

3.1.1 Loss functions

Choosing loss function is depend on the objective of problem that the NN is being trianed to to solve. For classification problems, such as digit classification, which the goal is to categorize $\boldsymbol{x}$ into K categories C, $f : \boldsymbol{x} \in \mathbb{R}^d \mapsto C \in \{C_k\}$, *Cross-Entropy*(CE) is the loss function for this purpose.

$$l_{\text{CE}} = -\sum_{i} y_k \log \hat{y}_k,$$

where $y_i \in [0, 1]$ and $\hat{y}_i \in [0, 1]$ are true and predicted probability that $\boldsymbol{x}$ belongs to C_k respectively. Consider a K-class classification problem, $\boldsymbol{z} = f(\boldsymbol{x}) \in \mathbb{R}^K$, $\hat{y}_k$ is computed via *softmax* activation function :

$$\hat{y}_k = \frac{e^{z_k}}{\sum_{k=1}^{K} e^{z_k}}$$

For regression problems, $f : \boldsymbol{x} \in \mathbb{R}^d \mapsto \mathbb{R}$, such as forecasting price, requires *Mean Squared Error*(MSE) loss function.

$$l_{\text{MSE}} = (f(\boldsymbol{x}) - y)^2$$

This is a brief introduction to loss functions widely used in machine learning. More loss functions do exist and are beyond scope of the thesis to cover.

3.1.2 Learning Algorithm : Gradient Descent and Backpropagation

Consider a function $L(\theta) = \theta^2$ on Figure 3.3 as a simplified version of loss function averaged over all $\boldsymbol{x}^{(\alpha)} \in \mathcal{D}$ from a NN with a parameter θ. In this case, $\hat{\theta}$ can trivially computed by solving

$$\frac{dL(\theta)}{d\theta} \stackrel{!}{=} 0 \tag{3.2}$$

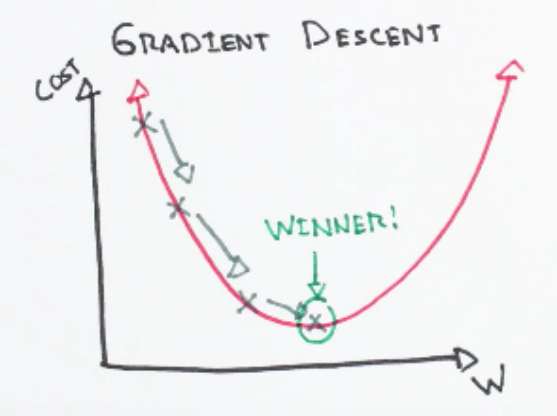


Figure 3.3: A Toy example

However, a NN usually contains thousands of parameters, hence $L(\boldsymbol{\theta})$ becomes a high dimensional function and solving (3.2) is not a trivial task. Nonetheless, Figure 3.3 shows an insight how we can reach the minimum location. There, we can see that if we move θ in the opposite direction of gradient $-\frac{dL(\theta)}{d\theta}$ with a proper step size λ (*learning rate*), we will eventually reach the minimum. Therefore, instead of directly solve (3.2), we rather slightly adjust $\boldsymbol{\theta}$ in the same direction of the gradient for some iterations, for example until the loss function converges. This is called *Gradient Descent*.

$$\forall \theta_i \in \boldsymbol{\theta} : \theta_i \leftarrow \theta_i - \frac{\lambda}{p} \sum_{\alpha=1}^{p} \frac{\partial l(f(\boldsymbol{x}^{(\alpha)}), y^{(\alpha)})}{\partial \theta_i}$$

Let's further consider a simple NN shown in Figure 3.4 with $\boldsymbol{\theta} = \{\forall i, j, k, l : w_{i\to j}^{(1)}, w_{j\to k}^{(2)}, w_{k\to l}^{(3)}\}$ and biases are omitted. $f(\boldsymbol{x}, \boldsymbol{\theta})$ is calculated as follows

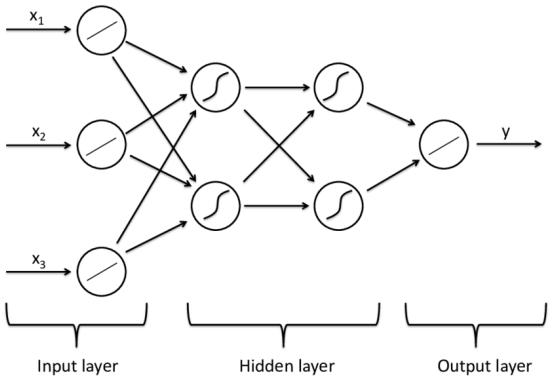


Figure 3.4: A Toy example

$$h_j^{(1)} = \sum_i w_{i\to j}^{(1)} x_i \qquad a_j^{(1)} = \sigma(h_j^{(1)})$$
$$h_k^{(2)} = \sum_j w_{j\to k}^{(2)} a_j^{(1)} \qquad a_k^{(2)} = \sigma(h_k^{(2)})$$
$$h_l^{(3)} = \sum_k w_{k\to l}^{(3)} a_k^{(2)} \qquad a_l^{(3)} = \sigma(h_l^{(3)})$$
$$f(\boldsymbol{x}) = [a_1^{(3)}, \ldots, a_L^{(3)}]^T \qquad L = \frac{1}{p}\sum_{\alpha=1}^{p} l(f(\boldsymbol{x}^{(\alpha)}), y^{(\alpha)})$$

Consider a sample $(\boldsymbol{x}^{(\alpha)}, ya)$. The gradients can be computed by recursively applying

chain rule to l from the last to the first layer, hence the name *Backpropagation*.

$$\frac{\partial l(f(\boldsymbol{x}^{(\alpha)}), y^{(\alpha)})}{\partial w^{(3)}_{k\to l}} = \frac{\partial l(f(\boldsymbol{x}^{(\alpha)}), y^{(\alpha)})}{\partial a^{(3)}_{l}} \frac{\partial a^{(3)}_{l}}{\partial w^{(3)}_{k\to l}} \tag{3.3}$$

$$= \underbrace{\frac{\partial l(f(\boldsymbol{x}^{(\alpha)}), y^{(\alpha)})}{\partial a^{(3)}_{l}} \sigma'(h^{(3)}_{l})}_{\delta^{(3)}_{l}} a^{(2)}_{k} \tag{3.4}$$

$$\frac{\partial l(f(\boldsymbol{x}^{(\alpha)}), y^{(\alpha)})}{\partial w^{(2)}_{j\to k}} = \sum_{l'=1}^{L} \frac{\partial l(f(\boldsymbol{x}^{(\alpha)}), y^{(\alpha)})}{\partial a^{(3)}_{l'}} \frac{\partial a^{(3)}_{l'}}{\partial w^{(2)}_{j\to k}} \tag{3.5}$$

$$= \sum_{l'=1}^{L} \frac{\partial l(f(\boldsymbol{x}^{(\alpha)}), y^{(\alpha)})}{\partial a^{(3)}_{l'}} \sigma'(h^{(3)}_{l'}) \frac{\partial h^{(3)}_{l'}}{\partial w^{(2)}_{j\to k}} \tag{3.6}$$

$$= \sum_{l'=1}^{L} \delta^{(3)}_{l'} w^{(3)}_{k\to l'} \frac{\partial a^{(2)}_{k}}{\partial w^{(2)}_{j\to k}} \tag{3.7}$$

$$= a^{(1)}_{j} \underbrace{\sigma'(h^{(2)}_{k}) \sum_{l'=1}^{L} \delta^{(3)}_{l'} w^{(3)}_{k\to l'}}_{\delta^{(2)}_{k}} \tag{3.8}$$

$$\frac{\partial l(f(\boldsymbol{x}^{(\alpha)}), y^{(\alpha)})}{\partial w^{(1)}_{i\to j}} = x_i \sigma'(h^{(1)}_{j}) \sum_{k'=1}^{K} \delta^{(2)}_{k'} w^{(2)}_{j\to k'} \tag{3.9}$$

As shown in the derivation, *Backpropagation* allows us to efficiently compute the gradients by reusing calculated quantities from computation of later layer, for example $\delta^{(3)}_{l}, \delta^{(2)}_{k}$. Moreover, $\delta^{(3)}_{l}, \delta^{(2)}_{k}$ can interpreted as error of the neuron.

In practice, because the training set usually contains several thousand samples, the gradient update in (3.3) would require significant amount of computation to update one step, not to mention that it could also result in small gradient update step leading to slow convergence to desire objective performance. Therefore, the training data D is usually divided into batches $\widetilde{D}_i$ with equal size and perform the gradient update for every $\widetilde{D}_i$. For example, the size of $\widetilde{D}_i$ is usually chosen between 32 and 512 samples. This refers to *Mini-Batch Gradient Descent*.

Lastly, because noise in the data and potentially highly non-smooth of the loss function, learning rate λ is a great influential to the training process. More precisely, it should not be too small or too large. This requires some effort and experience in order to get the value right. Some work have proposed alternative update rules aiming to make the training process more stable. For example, Adaptive Moment Estimation(Adam)[Kingma and Ba, 2014] uses adaptive learning rate and incorporates accumulated direction and speed of the previous gradients (momentum) into the update, hence more consistent gradient and fast convergence. Other similar proposals are RMSProp [Tieleman and Hinton, 2012] and Adadelta [Zeiler, 2012].

3.1.3 Convolutional Neural Networks

Convolutional Neural Networks(CNNs) refer to neural networks that employ convolutional operators to process information in early layers instead of fully-connected layers(weighted sum). Typically, a convolutional operator is followed by a pooling operator. Using this convolutional and pooling operators allows the NN to extract hierarchical features that are spatially invariant [Zeiler and Fergus, 2013], hence having higher predictive capability of the NN comparing to traditional fully-connected layers with the same number of parameters.

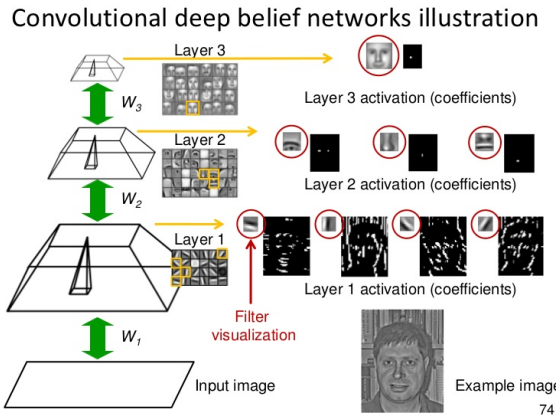


Figure 3.5: An intuition of what features CNN learns

Figure 3.5 illustrates an intuition behind hierarchical structures that CNN's neurons learn to detect. More precisely, in this example, neurons in the first learn to detect low level features, such as edges, and neurons in middle layer then use knowledge to detect higher level features, for example nose, mouth or eyes, and so on.

Since [LeCun et al., 2001] successfully applied CNNs to a document classification problem, CNNs have become the first choice of architectures in many domains. Particularly, in computer vision, CNNs are the core component of state-of-the-art results in various contests. Such successful results are : AlexNet[Krizhevsky et al., 2012] that archive the remarkable results on ImageNet Large-Scale Visual Recognition Challenge 2012(ILSVRC 2012) followed by the achievement of VGG[Simonyan and Zisserman, 2014] and GoogleLenet [Szegedy et al., 2014] architecture in ILSVRC 2014 and ResNet[He et al., 2015] that won ILSVRC 2015.

3.1.4 Recurrent Neural Networks

Recurrent Neural Networks(RNNs) are neural networks whose computed outputs are repeatedly incorporated into the next computation. Figure 3.6 illustrates this idea of recurrent computation by unfolding RNN into steps. Let's consider $\boldsymbol{x}$ a sequence of $x_1, \dots x_t$. At step t, RNN takes r_{t-1} and x_t to compute r_t and $\hat{y}_t$. This recurrent

connections can be interpreted as accumulating information from the past, hence RNNs are capable of processing sequential data, possibly coming with different length. Natural Language Processing(NLP) and Machine Translation(MT) are some of the fields that RNNs are widely applied to.

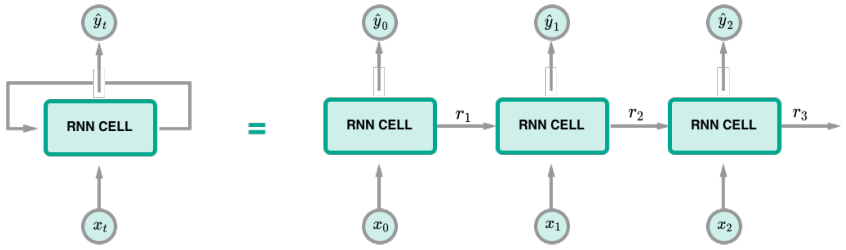


Figure 3.6: RNN Structure

Backpropagation Through Time

As the number of computation steps in RNNs is depend on the length of samples, which can be different in principle, one needs to organize data in such a way that samples in the same batch have the same steps of computations before training a RNN. In this case, training RNNs can be viewed as training a feedforward neural network where we apply backpropagation, except that variables are shared across steps.

Consider again the RNN in Figure 3.6 with $\boldsymbol{x} = \{x_1, \ldots, x_T\}$ and $r_0 = 0$. Assume that only $\hat{y}_T$ determines the value of the loss function and the computations are defined as follows

$$h_1 = w_{rx}x_1 + w_{rr}r_0 \qquad r_1 = \sigma(h_1) \tag{3.10}$$
$$h_2 = w_{rx}x_2 + w_{rr}r_1 \qquad r_2 = \sigma(h_2) \tag{3.11}$$
$$\vdots \qquad \vdots \tag{3.12}$$
$$h_{T-1} = w_{rx}x_{T-1} + w_{rr}r_{T-2} \qquad r_{T-1} = \sigma(h_{T-1}) \tag{3.13}$$
$$\hat{y} = \sigma(w_{yx}x_T + w_{yr}r_{T-1}) \tag{3.14}$$

The gradients can be computed by

$$\frac{\partial l}{\partial w_{yx}} = \sigma'(w_{yx}x_T + w_{yr}r_{T-1})x_T \tag{3.15}$$

$$\frac{\partial l}{\partial w_{yr}} = \sigma'(w_{yx}x_T + w_{yr}r_{T-1})r_{T-1} \tag{3.16}$$

$$\frac{\partial l}{\partial w_{rx}} = w_{yr}\sigma'(w_{yx}x_T + w_{yr}r_{T-1})\frac{\partial r_{T-1}}{\partial w_{rx}} \tag{3.17}$$

$$= w_{yr}\sigma'(w_{yx}x_T + w_{yr}r_{T-1})\left[\sigma'(h_{T-1})\left(x_{T-1} + w_{rr}\frac{\partial r_{T-2}}{\partial w_{rx}}\right)\right] \tag{3.18}$$

$$\frac{\partial l}{\partial w_{rr}} = w_{yr}\sigma'(w_{yx}x_T + w_{yr}r_{T-1})\frac{\partial r_{T-1}}{\partial w_{rr}} \tag{3.19}$$

$$= w_{yr}\sigma'(w_{yx}x_T + w_{yr}r_{T-1})\left[\sigma'(h_{T-1})\left(\frac{\partial r_{T-2}}{\partial w_{rr}}\right)\right] \tag{3.20}$$

$$\tag{3.21}$$

However, as we unfold the computations, we can see that there are 2 possibilities that might happen to the gradients of the shared parameters w_{rx} and w_{rr}, namely

- Exploding Gradient : this scenario happens if the gradient is derived from shared weights, for example w_{rr} in (3.18), whose absolute value is greater than one. The recursive multiplication will result in a large value of the gradient leading to unreliable training. [Pascanu et al., 2012] have proposed Gradient Clipping to alleviate the problem.
- Varnishing Gradient : in contrast, when the values are smaller than one, the gradient will be very small causing slow learning. More precisely, RNNs would require enormous of time to learn long term dependencies. The next section discusses techniques to mitigate this problem

Long Short-Term Memory and Gated RNNs

Varnishing Gradient is a major problem that causes RNNs to learn long term memories with slow progress. This is due to how the computation of recurrent connections are constructed. In particular, as in (3.10), standard RNNs compute those connections with weighted sum at every step t leading to recursive multiplication terms in the gradient's computation.

Alternatively,[Hochreiter and Schmidhuber, 1997] have proposed *Long Short-Term Memory*(LSTM) network that employs a gating mechanism and additive updates in the calculation of the recurrent connections. This mechanism decreases number of damping factors involved in the gradients' computation, hence it allows the network to learn long memories better.

More precisely, as shown in Figure 3.7, LSTM utilizes 3 gates, namely input i_g, forget f_g and o_g output gate, to control the information flow through the LSTM cell. In

particular, i_g and f_g decides how to accumulate information in the cell state C_t from previous cell state C_{t-1}, previous output h_{t-1} and current input x_t, while o_g determines to the information in C_t leaks to outside h_t. Formally,

$$i_g = \sigma(w_{ix}x_t + w_{ih}h_{t-1}) \qquad f_g = \sigma(w_{fx}x_t + w_{fh}h_{t-1}) \tag{3.22}$$

$$o_g = \sigma(w_{ox}x_t + w_{oh}h_{t-1}) \qquad \widetilde{C}_t = \tanh(w_{cx}x_t + w_{ch}h_t) \tag{3.23}$$

$$C_t = f_g \odot C_{t-1} + i_g \odot \widetilde{C}_t \qquad h_t = o_g \odot \tanh(C_t) \tag{3.24}$$

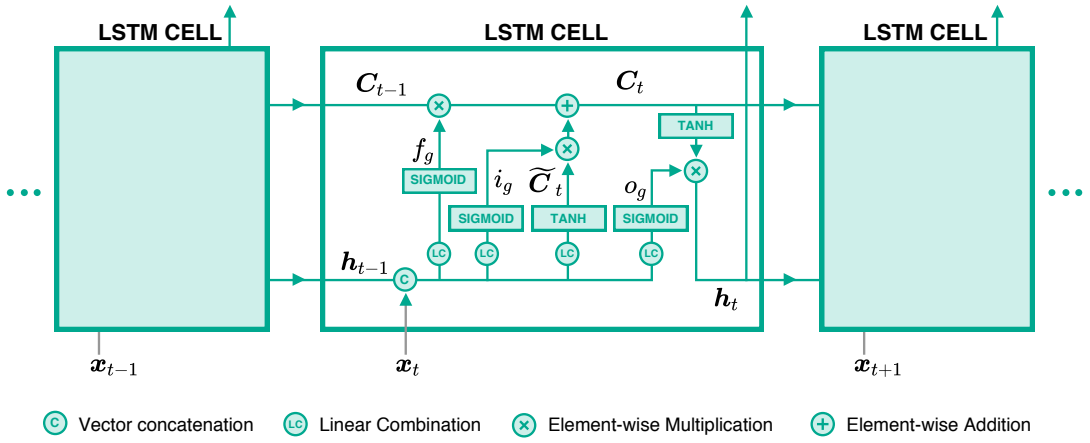


Figure 3.7: LSTM Structure

Since the work published, LSTM has successfully contributed to many state-of-the-art results, such as MT ..... [Greff et al., 2017] have shown that the forget and output gate are the crucial parts of the network. [Cho et al., 2014] have proposed *Gated Recurrent Unit*(GRU) that employs only 2 gates, however [Jozefowicz et al., 2015] have conducted several benchmarking tasks and found no significant difference in performance between LSTM and GRU.

3.2 Explainability of Neural Networks

Neural networks have become one of major machine learning algorithms used in many applications, for example computer vision and medicine. Despite those achievements, they are still considered as a blackbox process that is difficult to interpret its results or find evidences that the networks use to make such accurate decisions for further analysis.

Moreover, it is always important to verify whether the trained network properly utilize data or what information it uses to make decisions. Literatures usually call this process as "Explainability". [Bach et al., 2016] show that there are situations that the networks exploit artifacts in the data to make decisions. This discovery emphasizes the importance of having explainable neural networks, not to mention the fact that we are applying more and more neural networks to domains that human's life involved, such as medicine or self-driving car.

There are 2 approaches towards explaining neural network, namely *Global* and *Local* analysis. Given a classification problem of $\mathcal{C}$ classes classification problem and a trained network, global method aims to find an input $\boldsymbol{x}^*$ that is the most representative to a class $c \in \mathcal{C}$. "Activation Maximization[Erhan et al., 2010]" is such method.

$$\boldsymbol{x}^* = \underset{\boldsymbol{x}}{\operatorname{argmax}}\ \mathbb{P}(c|\boldsymbol{x}, \theta)$$

On the other hand, local analysis focuses on finding relevant information in $\boldsymbol{x}$ that causes the network predicting class c_i. For example, consider an image classification problems, we can interpret a pixel $x_i \in \boldsymbol{x}$ as a feature, this local analysis tries to find relevance score of each pixel in respect to the classification decision. This process usually results in heatmap intensity, often called relevance heatmap $R(\boldsymbol{x})$.

The difference between the 2 approaches can be analogously described by formulating questions as follows : Consider $\boldsymbol{x}$ is an image in category "car".

- Global analysis : "what does the usual car look like?"
- Local analysis : "which area in the image make it look like car?" wheels, windows?

In the following, I will leave content of global analysis aside and discuss only approaches in local analysis further. In particular, I will start with properties that literatures usually use to analysis goodness of a method. These properties can imply the quality of relevance heatmap. Then, I will discuss 3 methods, namely Sensitivity Analysis, Simple Taylor Decomposition and Layer-wise Relevance Propagation.

Consider $f(\boldsymbol{x})$ is an output from a neural network classifier that is corresponding to the class prediction, for example the value at the final layer before applying softmax function.

Definition 3.2.1. Conservation Property

$$\forall \boldsymbol{x} : f(\boldsymbol{x}) = \sum_i R_i$$

Sum of relevance score of each pixel x_i should equal to the total relevance that the network outputs.

Figure 3.8: Comparison between Global and Local Analysis

Definition 3.2.2. Positivity Property

Definition 3.2.3. Consistency

3.2.1 Sensitivity Analysis

Sensitivity analysis[Simonyan et al., 2013] is a local analysis that derives relevance R_i of pixel x_i, from the partial derivative of $f(\boldsymbol{x})$ respect to x_i. In particular, literature usually formulates the calculation as

$$R_i = \left(\frac{\partial f(\boldsymbol{x})}{\partial x_i}\right)^2$$

Hence,

$$\sum_i R_i = ||\nabla f(\boldsymbol{x})||^2$$

Although this technique can be easily implemented via automatic differentiation provided in modern deep learning frameworks, such as TensorFlow[Abadi et al., 2016], the derivation of $\sum_i R_i$ above implies that sensitivity analysis instead seeks to explain R_i from the aspect of variation magnitudes, not the actual relevance quantity.

3.2.2 Guided Backpropagation

Guided backpropagation is a extended version of sensitivity analysis where gradients are propagated in a controlled manner. It is designed specifically for neural network using piecewise linear activations. In particular, Springenberg et al.[Springenberg et al., 2014] reformulate the definition of ReLU function as:

$$\sigma(x) = x\mathbb{1}[x > 0],$$

where $\mathbb{1}[\cdot]$ is an indicator function. With the new formulation, [Springenberg et al., 2014] proposes a new derivative of a ReLU neuron j as:

$$\frac{\partial_* f(\boldsymbol{x})}{\partial a_j} = \mathbb{1}\left[a_j > 0\right]\mathbb{1}\left[\frac{\partial f(\boldsymbol{x})}{\partial a_j} > 0\right]\frac{\partial f(\boldsymbol{x})}{\partial a_j}$$

The 2 indicator functions control whether original gradients are propagated back, hence the name "Guided Backpropagation". Hence, the relevance score for x_i is:

$$R_i = \left(\frac{\partial_* f(\boldsymbol{x})}{\partial x_i}\right)^2$$

With this result, one can see that x_i is relevant to the problem if activations a_j that it supplies are active and positively contribute to $f(\boldsymbol{x})$.

3.2.3 Simple Taylor Decomposition

This method decomposes $f(\boldsymbol{x})$ into terms of relevance scores R_i via Taylor Decomposition. Formally,

$$f(\boldsymbol{x}) = f(\tilde{\boldsymbol{x}}) + \sum_i \underbrace{\left.\frac{\partial f}{\partial x_i}\right|_{x_i=\tilde{x}_i}(x_i - \tilde{x}_i)}_{R_i} + \zeta,$$

where ζ is the second and higher order terms of Taylor series and $\tilde{\boldsymbol{x}}$ is a root point where $f(\tilde{\boldsymbol{x}}) = 0$. To find such $\tilde{\boldsymbol{x}}$, one need to optimize :

$$\min_{\xi\in\mathcal{X}} ||\xi - \boldsymbol{x}||^2 \qquad \text{such that } f(\xi) = 0,$$

where $\mathcal{X}$ represents the input distribution. However, this optimization is time consuming and ξ might potentially be close to or diverge from $\boldsymbol{x}$ leading to non informative R_i.

Nonetheless, Montavon et al.[Montavon et al., 2017b] demonstrate that neural networks whose activations $\sigma(x)$ are piecewise linear functions with $\sigma(tx) = t\sigma(x), \forall t \geq 0$ property, for example a deep Rectified Linear Unit (ReLU) network without biases, $\tilde{\boldsymbol{x}}$ can be found in approximately the same flat region as $\boldsymbol{x}$, $\tilde{\boldsymbol{x}} = \lim_{\epsilon\to 0}\epsilon\boldsymbol{x}$, yielding

$$\left.\frac{\partial f(\boldsymbol{x})}{\partial x_i}\right|_{\boldsymbol{x}=\tilde{\boldsymbol{x}}} = \left.\frac{\partial f(\boldsymbol{x})}{\partial x_i}\right|_{\boldsymbol{x}=\boldsymbol{x}}$$

Hence, the decomposition can be simplified to :

$$f(\boldsymbol{x}) = \sum_i \underbrace{\left.\frac{\partial f(\boldsymbol{x})}{\partial x_i}\right|_{\boldsymbol{x}=\boldsymbol{x}} x_i}_{R_i}$$

Moreover, the result suggests the relationship between sensitivity analysis and Taylor decomposition. Specifically, x_i has high relevance score if x_i activates and its variation positively affects $f(x)$ and vice versa.

3.2.4 Layer-wise Relevance Propagation

The methods mentioned so far derive R_i directly from $f(\boldsymbol{x})$ and do not use important knowledge about the network itself, such as architecture or activation values. Alternatively, Bach et al.[Binder et al., 0906] propose Layer-wise Relevance Propagation(LRP) framework that leverages this known information to distribute relevance scores to x_i. In particular, LRP propagates relevance scores backward from layer to layer, similar to the back-propagation algorithm of gradient descent.

Consider the neural network illustrated in Figure 3.9. R_j and R_k are relevance score of neurons j, k in successive layers. [Binder et al., 0906] formulates the general form of relevance propagation as :

$$R_j = \sum_k \delta_{j \leftarrow k} R_k, \tag{3.25}$$

where $\delta_{j \leftarrow k}$ defines a proportion that R_k contributes to R_j. Consider further that activity a_k of neuron k is computed by

$$a_k = \sigma\bigg(\sum_j w_{jk} a_j + b_k \bigg),$$

where w_{jk}, b_k are the corresponding weight and bias between neuron j and k, and σ is a monotonic increasing activation function. [Binder et al., 0906] suggests

$$\delta_{j \leftarrow k} = \alpha \frac{a_j w_{jk}^+}{\sum_j a_j w_{jk}^+} - \beta \frac{a_j w_{jk}^-}{\sum_j a_j w_{jk}^-}, \tag{3.26}$$

where w_{jk}^+, w_{jk}^- are $\max(0, w_{jk})$, $\min(0, w_{jk})$, and α, β are parameters with $\alpha - \beta = 1$ condition. Together with Equation 3.25, [Binder et al., 0906] calls $\alpha\beta$-rule.

$$R_j = \sum_k \left(\alpha \frac{a_j w_{jk}^+}{\sum_j a_j w_{jk}^+} - \beta \frac{a_j w_{jk}^-}{\sum_j a_j w_{jk}^-} \right) R_k \tag{3.27}$$

This $\alpha\beta$-rule ensures that conservation property is satisfied as $f(\boldsymbol{x})$ is distributed through the network.

$$\sum_i R_i = \sum_j R_j = \sum_k R_k = f(\boldsymbol{x})$$

Moreover, if we rewrites $\alpha\beta$-rule as

$$R_j = \sum_k \frac{a_j w_{jk}^+}{\sum_j a_j w_{jk}^+} \hat{R}_k + \frac{a_j w_{jk}^-}{\sum_j a_j w_{jk}^-} \check{R}_k,$$

where $\hat{R}_k = \alpha R_k$ and $\check{R}_k = -\beta R_k$. Then, we can intuitively interpret this propagation as

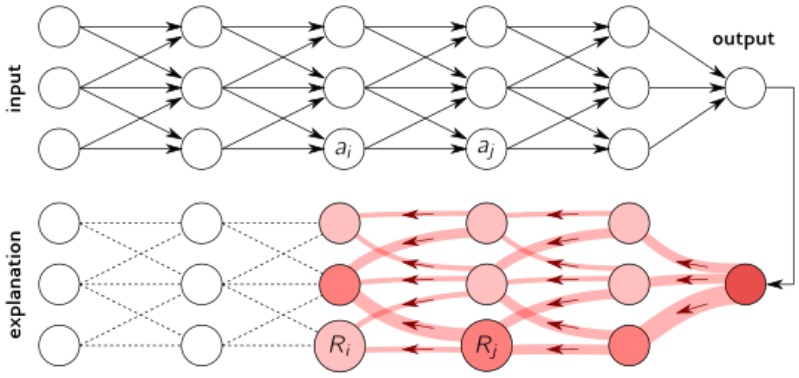


Figure 3.9: LRP Framework[2]

> "Relevance $\hat{R}_k$" should be redistributed to the lower- layer neurons $\{a_j\}_j$ in proportion to their excitatory effect on a_k. "Counter-relevance" $\check{R}_k$ should be redistributed to the lower-layer neurons $\{a_j\}_j$ in proportion to their inhibitory effect on a_j - Section 5.1 [Montavon et al., 2017b]

However, it seems that there is no clear relationship between values of α, β and the structure of the heatmap. [Montavon et al., 2017b, Binder et al., 0906] demonstrate that the values are depend on the architecture of the network. In particular, [Montavon et al., 2017b] observes that $\alpha = 1, \beta = 0$ works well for deep architectures, such as GoogleNet[Szegedy et al., 2014], while $\alpha = 2, \beta = 1$ for shallower architectures, such as BVLC CaffeNet[Jia et al., 2014].

Algorithm 1: LRP Algorithm

$f(\boldsymbol{x}), \{\{a\}_{l_1}, \{a\}_{l_2}, \ldots, \{a\}_{l_n}\} = \text{forward\_pass}(\boldsymbol{x}, \boldsymbol{\theta})$;
$R_k = f(\boldsymbol{x})$;
for layer $\in$ reverse$(\{l_1, l_2, \ldots, l_n\})$ **do**
  prev_layer $\leftarrow$ layer $- 1$;
  for $j \in$ neurons(prev_layer), $k \in$ neurons(layer) **do**
    $R_j \leftarrow \alpha\beta$-rule$(R_k, \{a\}_j, \{w\}_{j,k})$;
  end
end

3.2.5 Deep Taylor Decomposition

Deep Taylor Decomposition(DTD) is a explanation technique that decomposes R_k as a sum of R_j from previous layer using Simple Taylor Decomposition. Montavon et al.[Montavon et al., 2017a] proposes the method to explain decisions of neural networks

[2]Source: `http://heatmapping.org`

with piece-wise linear activations. Similar to LRP, DTD decomposes R_k and propagates the quantity backward to R_J. In particular, R_k is decomposed as follows :

$$R_k = R_k\Big|_{\tilde{\boldsymbol{a}}_j} + \sum_j \frac{\partial R_k}{\partial a_j}\Big|_{a_j=\tilde{a}_j} (a_j - \tilde{a}_j) + \zeta_k \tag{3.28}$$

Assume further that there exists a root point $\tilde{\boldsymbol{a}}_j$ such that $R_k = 0$, and the second and higher terms $\zeta_k = 0$. Then, Equation 3.28 is simplified to

$$R_k = \sum_j \underbrace{\frac{\partial R_k}{\partial a_j}\Big|_{a_j=\tilde{a}_j} (a_j - \tilde{a}_j)}_{R_{j\leftarrow k}} \tag{3.29}$$

Moreover, as the relevance propagated back, [Montavon et al., 2017a] shows that R_j is an aggregation of $R_{j\leftarrow k}$ from neuron k in the next layer that neuron j contributes to. Formally, this is

$$R_j = \sum_k R_{j\leftarrow k} \tag{3.30}$$

Combining Equation 3.29 and 3.30 yields :

$$\begin{aligned} R_j &= \sum_k R_{j\leftarrow k} \\ \sum_j R_j &= \sum_j \sum_k R_{j\leftarrow k} \\ \sum_j R_j &= \sum_k \sum_j R_{j\leftarrow k} \\ \sum_j R_j &= \sum_k R_k \end{aligned} \tag{3.31}$$

Demonstrated by [Montavon et al., 2017a], Equation 3.31 holds for all j, k and all subsequent layers. Hence, this results in conservation property which guarantee that no relevance loss during the propagations.

$$\sum_i R_i = \cdots = \sum_j R_j = \sum_k R_k = \cdots = f(\boldsymbol{x}) \tag{3.32}$$

To find a root point $\tilde{\boldsymbol{a}}_j$, consider a neural network whose R_k is computed by :

$$R_k = \max\left(0, \sum_j a_j w_{jk} + b_k\right), \tag{3.33}$$

where $b_k \leq 0$.

One can see that there are 2 cases to be analyzed, namely $R_k = 0$ and $R_k \geq 0$. For $R_k = 0$, $\boldsymbol{a}_j$ is already the root point. For the latter, one can find such point by performing line search in a direction $\boldsymbol{v}_j$ and magnitude t.

$$\tilde{a}_j = a_j - tv_j, \tag{3.34}$$

The root point is then the intersection point between Equation 3.34 and 3.33. Hence,

$$0 = \sum_j (a_j - tv_j)w_{jk} + b_k \tag{3.35}$$

$$t\sum_j v_j w_{jk} = R_k \tag{3.36}$$

$$t = \frac{R_k}{\sum_j v_j w_{jk}} \tag{3.37}$$

$$\tag{3.38}$$

Therefore, R_j can be computed by :

$$R_j = \sum_k \frac{\partial R_k}{\partial a_j}\bigg|_{a_j - \tilde{a}_j} (a_j - \tilde{a}_j) \tag{3.39}$$

$$= \sum_k w_{jk} t v_j \tag{3.40}$$

$$= \sum_k \frac{v_j w_{jk}}{\sum_j v_j w_{jk}} R_k \tag{3.41}$$

Noticing here is that $\tilde{\boldsymbol{a}}_j$ is not necessary the closest point to the line $R_k = 0$ in Euclidean distance, because $\tilde{\boldsymbol{a}}_j$ might not be in the same domain as $\boldsymbol{a}_j$, hence v_j needs to be chosen according to the domain of $\boldsymbol{a}_j$. Consider an example on Figure 3.10, if $a_j \in \mathbb{R}^+$, then $\widetilde{a_j}$ must be also in $\mathbb{R}^+$. In the following, I will summarize how $\boldsymbol{a}_j$ can be computed for each domain of $\boldsymbol{a}_j$.

Case $a_j \in \mathbb{R}$: w^2-rule

Trivially, the search direction v_j is just the direction of gradient $\frac{\partial R_k^{(l+1)}}{\partial a_j^{(l)}}$:

$$v_j = w_{jk}$$

Hence,

$$R_j = \sum_k \frac{w_{jk}^2}{\sum_j w_{jk}^2} R_k$$

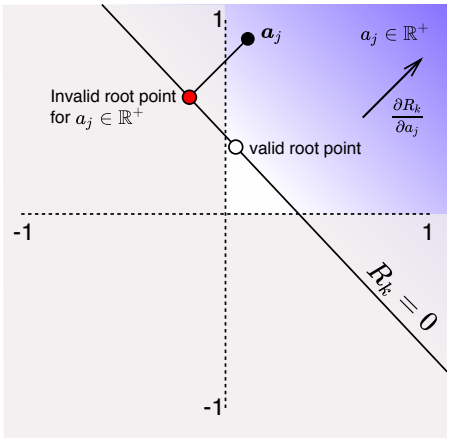


Figure 3.10: An illustration of R_k functional view and root point candidates

Case $a_j \geq 0$: z^+-rule

The root point is on the line segment $(\boldsymbol{a_j}\mathbb{1}\{w_{jk} < 0\}, \boldsymbol{a_j})$. In particular, as shown on Figure 3.11, R_k has a root at $\boldsymbol{a_j}\mathbb{1}\{w_{jk} < 0\}$, because of:

$$R_k = \max\left(\sum_j a_j w_{jk} + b_k, 0\right) \tag{3.42}$$

$$= \max\left(\sum_j a_j \mathbb{1}\{w_{jk} < 0\} w_{jk} + b_k, 0\right) \tag{3.43}$$

$$= \max\left(\sum_j a_j w_{jk}^- + b_k, 0\right) \tag{3.44}$$

$$= 0 \tag{3.45}$$

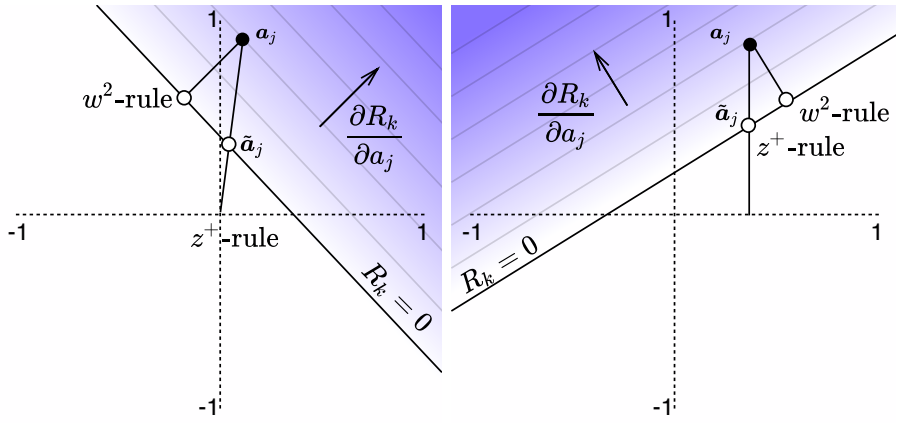
The last step uses the fact that $a_j \in R^+$ and $b_k \leq 0$ from the assumption. Hence, the search direction is:

$$\begin{aligned} v_j &= a_j - a_j \mathbb{1}\{w_{jk} < 0\} \\ &= a_j \mathbb{1}\{w_{jk} \geq 0\} \end{aligned}$$

Therefore,

$$\begin{aligned} R_j &= \sum_k \frac{w_{jk} a_j \mathbb{1}\{w_{jk} \geq 0\}}{\sum_j w_{jk} a_j \mathbb{1}\{w_{jk} \geq 0\}} R_k \\ &= \sum_k \frac{a_j w_{jk}^+}{\sum_j a_j w_{jk}^+} R_k \end{aligned}$$

Moreover, one can also see that z^+-rule is equivalent to LRP's $\alpha\beta$-rule when $\alpha = 1, \beta = 0$.



(a) (b)

Figure 3.11: R_k functional view and root points from z^+-rule

Case $l_j \leq a_j \leq h_j$ where $l_j \leq 0 < h_j$: z^β-rule

In this case, the root point is on the line segment $(l_j \mathbb{1}\{w_{jk} \geq 0\} + h_j \mathbb{1}\{w_{jk} \leq 0\}, a_j)$. One can show that

$$R_k = \max\left(\sum_j a_j w_{jk} + b_k, 0\right) \tag{3.46}$$

$$= \max\left(\sum_j (l_j \mathbb{1}\{w_{jk} \geq 0\} + h_j \mathbb{1}\{w_{jk} \leq 0\}) w_{jk} + b_k, 0\right) \tag{3.47}$$

$$= \max\left(\sum_j l_j w_{jk}^+ + h_j w_{jk}^- + b_k, 0\right) \tag{3.48}$$

$$= 0 \tag{3.49}$$

Hence, the search direction is

$$\begin{aligned} v_j &= a_j - \tilde{a}_j \\ &= a_j - l_j \mathbb{1}\{w_{jk} \geq 0\} - h_j \mathbb{1}\{w_{jk} \leq 0\} \end{aligned}$$

Figure 3.12 illustrates details of the search direction. Therefore,

$$\begin{aligned} R_j &= \sum_k \frac{w_{jk}(a_j - l_j \mathbb{1}\{w_{jk} \geq 0\} - h_j \mathbb{1}\{w_{jk} \leq 0\})}{\sum_j w_{jk}(a_j - l_j \mathbb{1}\{w_{jk} \geq 0\} - h_j \mathbb{1}\{w_{jk} \leq 0\})} R_k \\ &= \sum_k \frac{a_j w_{jk} - l_j w_{jk}^- - h_j w_{jk}^+}{\sum_j a_j w_{jk} - l_j w_{jk}^- - h_j w_{jk}^+ -} R_k \end{aligned}$$

In summary, DTD is a theory for explaining nonlinear compuations of neural network through decomposing relevance score between successive layers. Its propagation

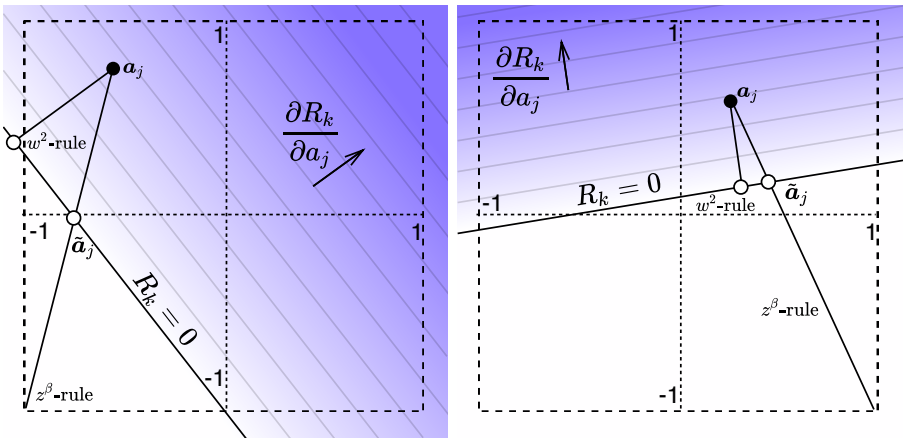


Figure 3.12: R_k functional view and root points from z^β-rule with $-1 < a_j < 1$

rules ensures conservation property. Given the rules above, the relevance scores can be propagated using LRP Algorithm 1.

Lastly, as DTD provides more general propagation rules than the $\alpha\beta$-rule from LRP, I will use DTD and LRP interchangeably throughout the thesis. In particular, it will be mentioned explicitly if $\alpha\beta$ is being used, otherwise the rule is a DTD's rule. Table 3.1 concludes the details when such DTD rules should be used.

Input Domain	LRP Propagation Rule
w^2-rule : Real values, $a_j \in R$	$R_j = \sum_k \frac{w_{jk}^2}{\sum_j w_{jk}^2} R_k$
z^+-rule : ReLU activations, $a_j \geq 0$	$R_j = \sum_k \frac{a_j w_{jk}^+}{\sum_j a_j w_{jk}^+} R_k$
z^β-rule : Pixel Intensities, $l_j \leq a_j \leq h_j$, $l_j \leq 0 \leq h_j$	$R_j = \sum_k \frac{a_j w_{jk} - l_j w_{jk}^- - h_j w_{jk}^+}{\sum_j a_j w_{jk} - l_j w_{jk}^- - h_j w_{jk}^+ -} R_k$

Table 3.1: LRP rules from Deep Taylor Decomposition

4 Experiments

4.1 Dataset

4.1.1 MNIST

MNIST[LeCun and Cortes, 2010] is one of the most popular dataset that machine learning partitioners use to benchmark machine learning algorithms. The dataset consists of 60,000 training and 10,000 testing samples. Each sample is a grayscale 28x28 image of a digit between from 0 to 9.

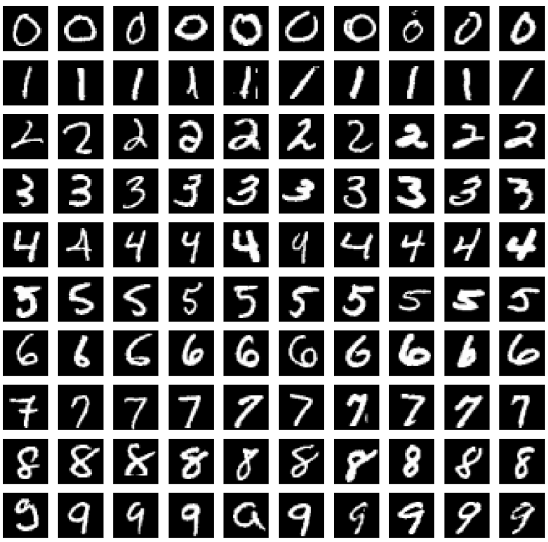


Figure 4.1: MNIST Dataset

State-of-the-art algorithms can classify MNIST with accuracy higher than 0.99, while classical ones, such as SVC or RandomForest, are able to achieve around 0.97[Xiao et al., 2017].

4.1.2 Fashion-MNIST

Xiao et. al.[Xiao et al., 2017] propose a novel dataset, called Fashion-MNIST dataset, as a replacement of MNIST dataset for benchmarking machine learning algorithms. According to [Xiao et al., 2017], Fashion-MNIST brings more challenging to the problem and more representative to modern computer vision tasks. It contains images of fashion products from 10 categories. Fashion-MNIST is comparable to MNIST in every aspects, such as the size of training and testing set, image dimension and data format, hence one

can easily apply existing algorithms that work with MNIST to Fashion-MNIST without any change.

Figure 4.2: Fashion-MNIST DatasetTODO : Figure use same size as MNIST figure

[Xiao et al., 2017] also reports benchmarking results of classical machine learning algorithms on Fashion-MNIST. On average, they achieve accuracy between 0.85 to 0.89. According to Fashion-MNIST's page[1], A. Brock reports the state-of-the-art result with 0.97 accuracy using Wide Residual Network(WRN)[Zagoruyko and Komodakis, 2016] and standard data preprocessing and augmentation.

4.2 RNN Cell Architectures

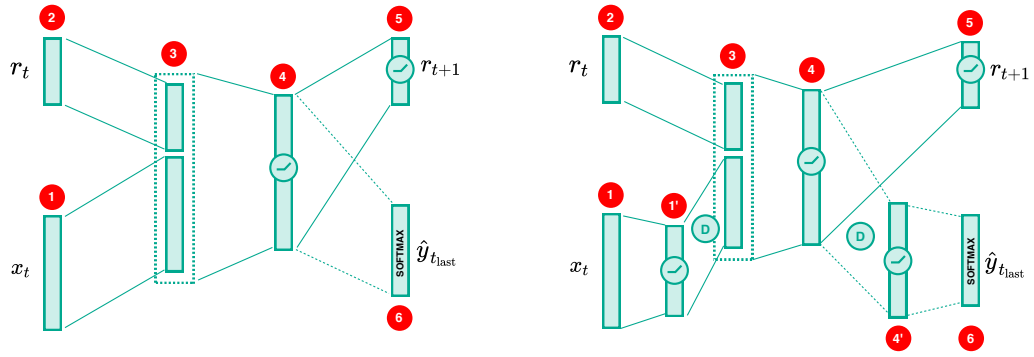
In this section, I will describe architectures or RNN cell evaluated in this thesis. To make the descriptions concise, I denote notations as follows:

- $\boldsymbol{a}_t^{(l)}$: activation vector of layer l at step t

[1]https://github.com/zalandoresearch/fashion-mnist

- $\boldsymbol{x}_t$: subset of $x_i \in \boldsymbol{x}$ corresponding to step t and assume to be reshaped into a column vector

4.2.1 Shallow Cell



(a) Shallow cell (b) Deep cell

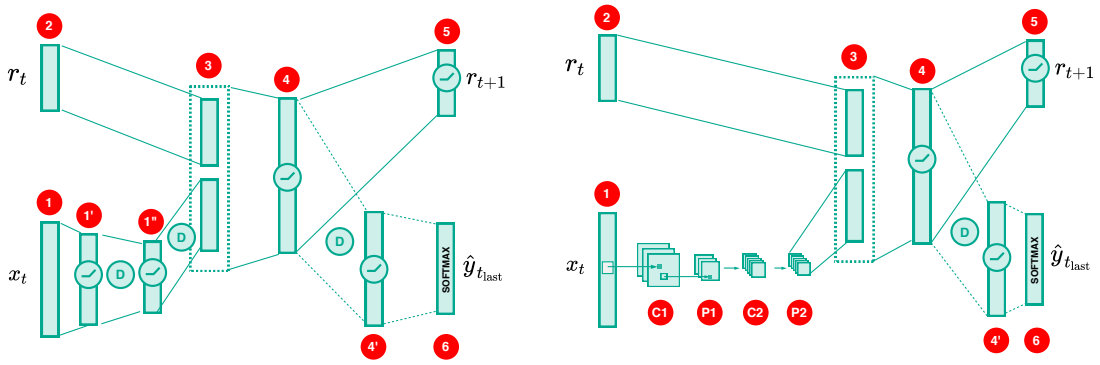
Figure 4.3: Shallow and Deep Cell Architecture

As shown in Figure 4.3a, Shallow cell first concatenates input $\boldsymbol{x}_t$ (1) and recurrent input $\boldsymbol{r}_t$ (2) at layer (3) as one vector before computing $\boldsymbol{a}_t^{(4)}$ of layer (4). Then, the next recurrent input $\boldsymbol{r}_{t+1}$ (5) is derived from $\boldsymbol{a}_t^{(4)}$. In the last step t_{last}, the raw output $\boldsymbol{h}$ is computed from $\boldsymbol{a}_{t_{\text{last}}}^{(4)}$ and applied to softmax function to compute class probabilities $\hat{\boldsymbol{y}}$ (6).

4.2.2 Deep Cell Architecture

Figure 4.3b illustrates the architecture of Deep cell. It can be viewed as an extension of the Shallow cell with 2 additional layers, namely (1') and (4'). The ideas of introducing the layers are to let (1') learn representations of the problem, while (4) can focus on combining information from past and current step, which enables (4') to compute more fine-grained decision probabilities. Dropout is applied at $\boldsymbol{a}^{(1')}$, and $\boldsymbol{a}_{t_{\text{last}}}^{(4)}$ for computing $\boldsymbol{a}^{(4')}$.

Two variations of Deep cell are also experimented, namely DeepV2 and ConvDeep, shown on Figure 4.4. The former has one additional layer (1'') with dropout regularization between (1'). On the other hand, the latter replaces fully connected layers between (1) and (3) with 2 convolutional and max pooling layers, [(C1), (P1)] and [(C2), (P2)].



(a) DeepV2 (b) ConvDeep

Figure 4.4: DeepV2 and ConvDeep Cell Architecture

4.3 General Setting

I implemented experiments using TensorFlow[2]. weights $w_{ij} \in \boldsymbol{W}$ and biases $b_j \in \boldsymbol{b}$ are initialized as follows:

$$w_{ij} \sim \Psi(\mu = 0, \sigma = 0.1, [-2\sigma, 2\sigma])$$
$$b_j = \ln(e^{0.01} - 1)$$

where $\Psi(\cdot)$ denotes Truncated Normal Distribution with valid region between $[-2\sigma, 2\sigma]$.

Activations $\boldsymbol{a}^{(l)}$ of layer l is calculated using :

$$\boldsymbol{h}^{(l)} = \boldsymbol{W}^T \boldsymbol{a}^{(l-1)} - \sigma_s(\boldsymbol{b})$$
$$\boldsymbol{a}^{(l)} = \sigma_r(\boldsymbol{h}^{(l)})$$

where

$$\sigma_r(h_j) = \max(0, h_j) \qquad \text{(ReLU function)}$$
$$\sigma_s(b_j) = \log(1 + \exp b_j) \qquad \text{(Softplus function)}$$

The reason of using $\sigma_s(b_j)$ for bias term is due to the non positive bias assumption of DTD. Moreover, $\sigma_s'(b_j)$ is $(0, \infty)$, as a result the network has more flexibility to adjust decision through back-propagation rather than using $\sigma_r(b_j)$. With this setting, the initial value of bias term $\sigma_s(b_j)$ is then 0.01.

Speaking about training methodology, I use Adam[Kingma and Ba, 2014] optimizer to train networks. Preliminary study shows that relevance heatmaps from networks trained by Adam are less noisy than the ones from other optimizers, such as Stochastic Gradient Descent(SGD). Number of epochs is set to 200, while dropout probability is

[2] `http://tensorflow.org/`

0.5 and batch size is 50. Learning rate is not fixed as it is likely that different network architectures requires different value to achieve good performance, hence left adjustable. Table 4.1 summaries the setting of hyperparameters.

Hyperparameter	**Value**
Optimizer	Adam
Epoch	200
Dropout Probability	0.5
Batch size	50

Table 4.1: Hyperparameter Summary

Traditionally, number of neurons in each layer ($n^{(l)}$) is another hyperparameter that we can adjust. However, as the goal is to compare relevance heatmaps from different architectures, those numbers are fixed and chosen in such a way that total number of variables in each architecture are equivalent. Figure 4.5 illustrates the details of the settings.

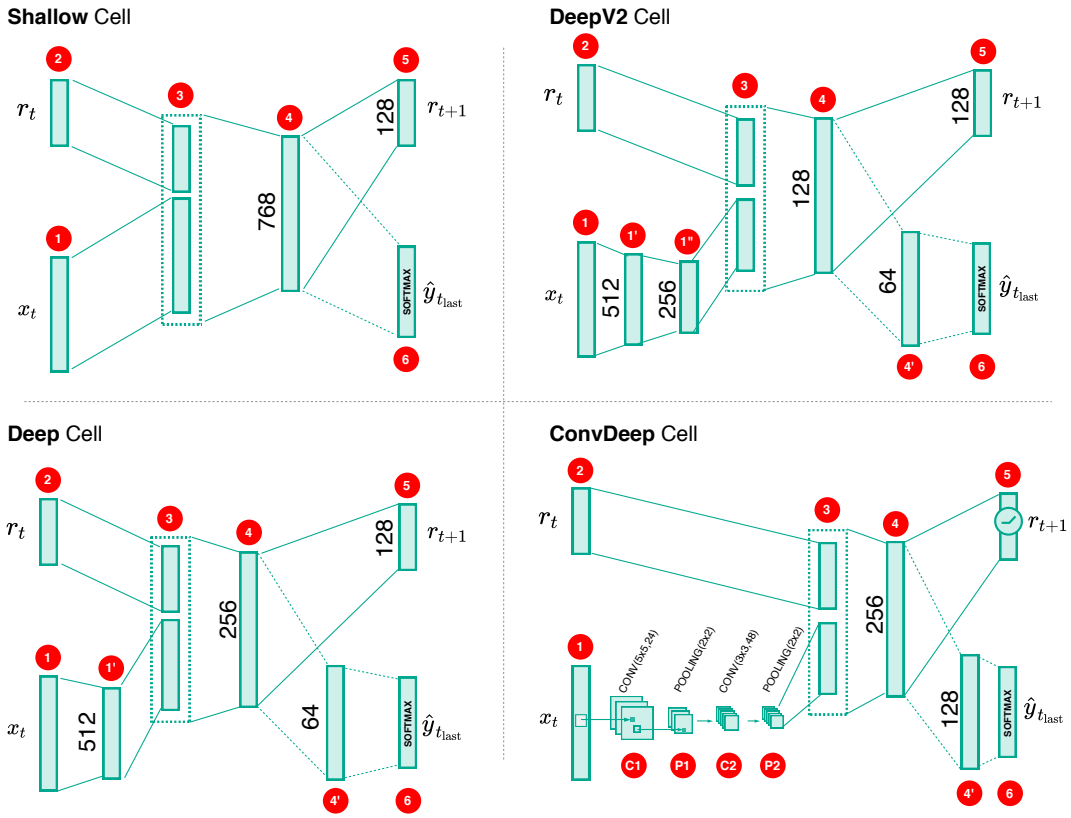


Figure 4.5: Number of neurons in each layer for each cell architecture

- **Shallow Cell**

$$\{n^{(4)}\} = \{768\}$$

- **Deep Cell**

$$\{n^{(1')}, n^{(4)}, n^{(4')}\} = \{512, 256, 64\}$$

- **DeepV2 Cell**

$$\{n^{(1')}, n^{(1")}, n^{(4)}, n^{(4')}\} = \{512, 256, 128, 64\}$$

- **ConvDeep Cell** :

$$\{n^{(C1)}, n^{(P1)}\} = \{CONV(5\text{x}5, 24), POOL(2\text{x}2)\}$$
$$\{n^{(C2)}, n^{(P2)}\} = \{CONV(3\text{x}3, 48), POOL(2\text{x}2)\}$$
$$\{n^{(4)}, n^{(4')}\} = \{256, 128\}$$

  where $CONV(x, y)$ is a convolutional operator with y filters whose kernel size is $\mathbb{R}^x$. Similarly, $POOL(x)$ is a pooling operator with kernel size $\mathbb{R}^x$.

Noting that, $n^{(5)}$ is set at 128 for all architectures and 0 when the sequence length of the problem is 1. $n^{(6)}$ is equal to the number of categories of a problem, for example $n^{(6)} = 10$ MNIST. Table 4.2 shows the total numbers of variables in details.

	Sequence Length			
Cell Architecture	1	4	7	14
Shallow	610570	355722	291210	248202
Deep	550346	314954	271946	243274
DeepV2	575050	306890	263882	235210
ConvDeep	647594	283178	197162	197162

Table 4.2: Total variables in each architecture and sequence length

Lastly, as the quality of relevance heatmap depending on performance of the model, the minimum classification accuracy is set as in Table 4.3.

Dataset	**Minimum Accuracy**
MNIST	0.98
Fashion-MNIST	0.85

Table 4.3: Classification Accuracy Criteria

4.4 Experiment 1 : Sequence Classification

4.4.1 Problem Formulation

To demonstrate how well RNNs can distribute relevant quantities to input space, I formulated an artificial classification problem in which each image sample $\boldsymbol{x}$ is column-wise split into non-overlapping $(\boldsymbol{x}_t)_{t=1}^{T}$. The RNN classifier needs to summarize information from the sequence $(\boldsymbol{x}_t)_{t=1}^{T}$ to answer what is the class of $\boldsymbol{x}$.

Figure 4.6 illustrates the setting. Here, a MNIST sample $\boldsymbol{x} \in \mathbb{R}^{28,28}$ is divided to a sequence of $(\boldsymbol{x}_t \in \mathbb{R}^{28,7})_{t=1}^{4}$. At time step t, $\boldsymbol{x}_t$ is presented to the RNN classifier which yields recurrent input $\boldsymbol{r}_{t+1}$ for the next step. For the last step T, in this example $T = 4$, the RNN classifier computes $f(\boldsymbol{x}) \in \mathbb{R}^{10}$ and the class that $\boldsymbol{x}$ belongs to.

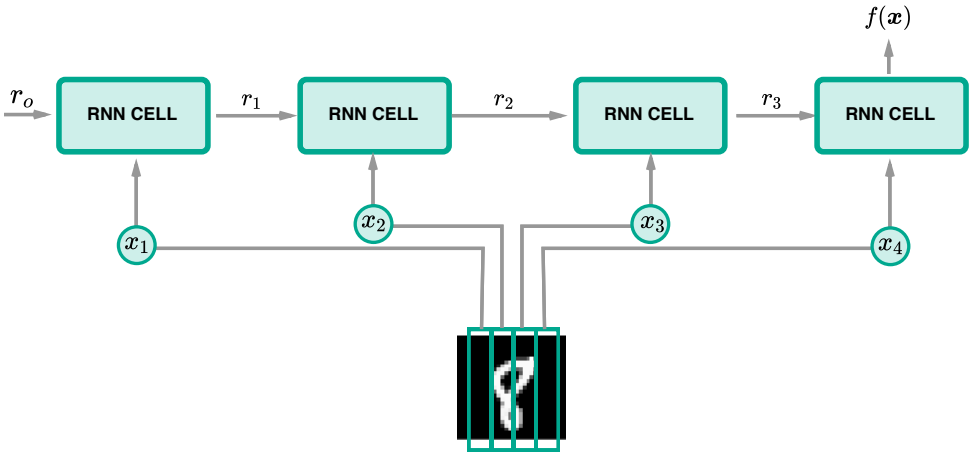


Figure 4.6: General setting of RNN classifiers in this thesis

Denote g_r and g_f function that the RNN with parameters $\boldsymbol{\theta}$ uses to compute $\boldsymbol{r}_{t+1}$ and $f(\boldsymbol{x})$ respectively. The overall computation can be summarized as follows:

$$\begin{aligned} \boldsymbol{r}_{t+1} &= g_r(\boldsymbol{\theta}, \boldsymbol{x_t}, \boldsymbol{r_t}) \\ &\vdots \\ f(\boldsymbol{x}) &= g_h(\boldsymbol{\theta}, \boldsymbol{x}_{t_{\text{last}}}, \boldsymbol{r}_{t_{\text{last}}}) \\ \hat{\boldsymbol{y}} &= \text{softmax}(f(\boldsymbol{x})), \end{aligned}$$

where $\hat{\boldsymbol{y}}$ is the class probabilities and $\boldsymbol{r}_0 = \boldsymbol{0}$. To explain the model, $R(\boldsymbol{x})$ is set to the value of $f(\boldsymbol{x})$ that is corresponding to the true target class.

TODO : Figure figure show how R(x) is set?

4.4.2 Result

I began this preliminary experiment with Shallow and Deep architecture. They were trained on MNIST and FashionMNIST with sequence length $T = \{1, 4, 7\}$. Table 4.4

	Sequence Length		
Dataset	1	4	7
MNIST / FashionMNIST	$\mathbb{R}^{28,28}$	$\mathbb{R}^{28,7}$	$\mathbb{R}^{28,4}$

Table 4.4: Dimensions of $\boldsymbol{x}_t$ for each dataset and sequence length

	Shallow	**Deep**
SEQ-1	xx.xx%	xx.xx%
SEQ-4	xx.xx%	xx.xx%
SEQ-7	xx.xx%	xx.xx%

Table 4.5: Model Accuracy

shows dimensions of $\boldsymbol{x}_t$ for different sequence length and Table 4.5 summaries accuracy of the trained models. To simplify the manuscript, I am going to use *ARCHITECTURE-T* convention to denote a RNN with *ARCHITECTURE* trained on sequence length *T*. For example, Deep-7 refers to the Deep RNN architecture trained on $(\boldsymbol{x}_t \in \mathbb{R}^{28,4})_{t=1}^{7}$.

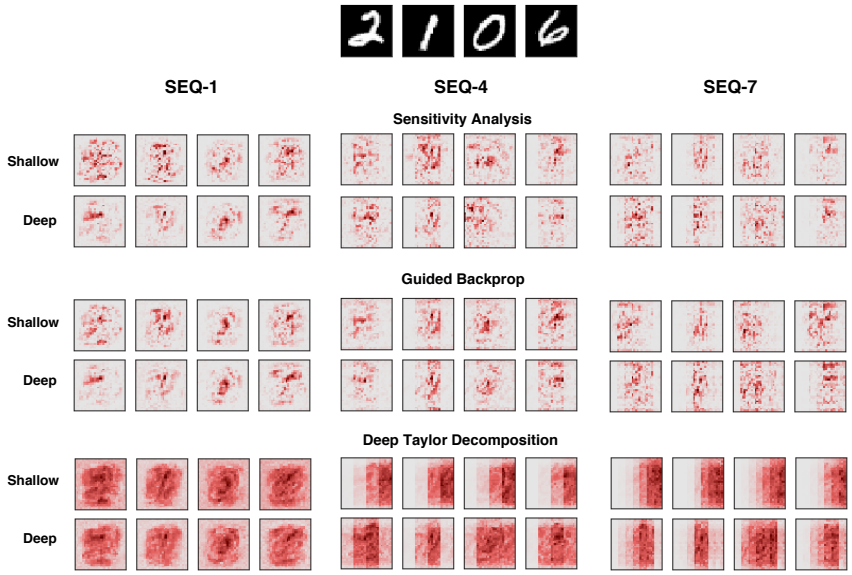


Figure 4.7: Relevance heatmaps from Shallow and Deep Cell trained on MNIST with different sequence lengths

Figure 4.7 shows relevance heatmaps from Shallow and Deep architecture trained on MNIST. We can see general characteristics of each explanation technique. In particular, sensitivity analysis(SA) and guided backprop(GB) heatmaps are sparse, while the ones from deep Taylor decomposition(DTD) are more diffuse throughout $\boldsymbol{x}$. When applying

these techniques to Shallow-1 and Deep-1, the relevance heatmaps look similar regardless of the architectures. As the sequence length is increased, SA and GB heatmaps are still almost identical for Shallow-4,7 and Deep-4,7. However, this is not the case for DTD. From the figure, we can see that Shallow-4,7 and Deep-4,7 return significantly different relevance heatmaps from DTD method. In particular, Shallow-4,7 's heatmaps are mainly concentrated on the right part of $\boldsymbol{x}$ associating to last time steps, while Deep-4,7's ones are appropriately highlighted at the actual content area of $\boldsymbol{x}$.

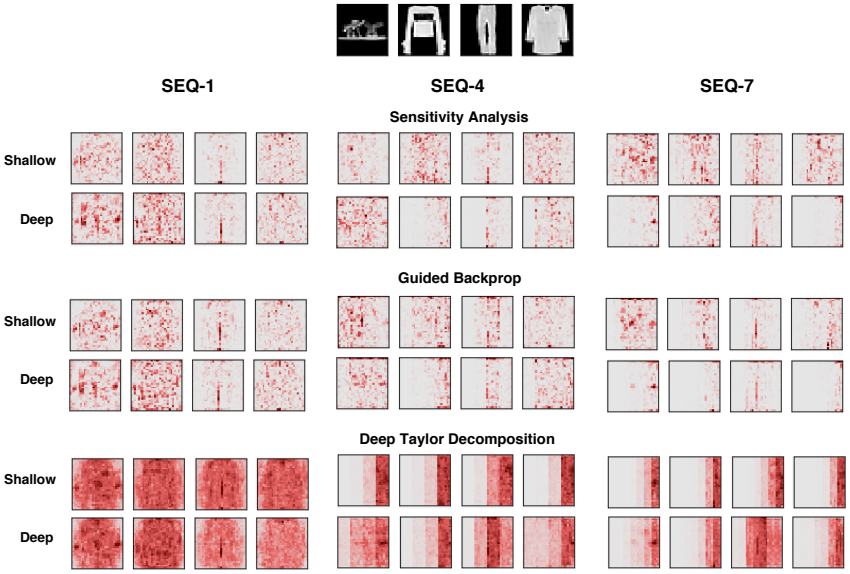


Figure 4.8: Relevance heatmaps from Shallow and Deep Cell trained on FashionMNIST with different sequence lengths

Relevance heatmaps of Shallow and Deep architecture trained on FashionMNIST are shown on Figure 4.8. Similar to MNIST, we do not see any remarkable difference on SA and GB heatmaps of the two architectures : only that Deep-4,7 produces slightly more sparse heatmaps. Although the wrong concentration issue of DTD seems to appear on both Shallow-4,7's and Deep-4,7's heatmaps, we still can observe proper highlight from Deep architecture on some samples. For example, the trouser sample, we can see that Deep-4,7 architecture manage to distribute high relevance scores to area of the trouser. Latent features of FashionMNIST might be one of the reasons why Deep architecture does not distribute relevance scores to early steps for the FashionMNIST samples. Consider *Shoe* and *Ankle Boot* samples in Figure 4.2. One can see that their front part are similar and only the heel part that determines the differences between the two categories.

Figure 4.9 presents relevance heatmaps of MNIST *Class 1* and FashionMNIST *Class Trouser* samples. These samples were chosen to emphasize the impact of RNN architecture on DTD explanation. In particular, these samples have $\boldsymbol{x}_{t'}$ containing features primarily locating at the center, or middle of the sequence, hence appropriate relevance heatmaps should be highlighted at $\boldsymbol{x}_{t'}$ and possibly its neighbors. As expected, we

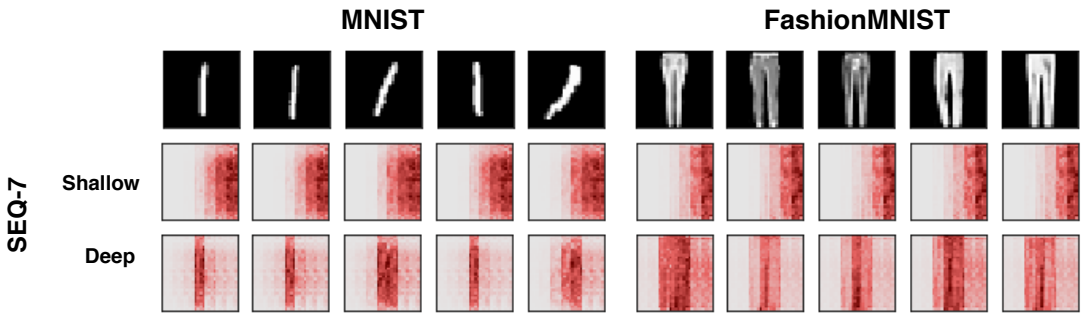


Figure 4.9: DTD relevance heatmaps of MNIST *Class 1* and FashionMNIST *Class Trouser* samples from Shallow-7 and Deep-7.

can see Deep-7 produces sound results in which the heatmaps have high intensity value where $\boldsymbol{x}_{t'}$ approximately locate, while Shallow-7 mainly assigns relevance quantities to $\boldsymbol{x}_t$ for $t \approx T$. Figure 4.10 further shows a quantitive evidence of this problem. Here, distribution of relevance score from Shallow-7 and Deep-7 are plotted across time step $t = \{1, \ldots, 7\}$. The distributions are computed from all test samples in MNIST *Class 1* and FashionMNIST *Class Trouser* respectively as well as the data distributions that are derived from pixel intensity values. We can see that Deep-7's relevance distributions are similar to the data distributions, while Shallow-7's ones diverge completely. Particularly, one can see that Shallow-7 distributes more than 90% of relevance scores to the last 3 steps, namely $\boldsymbol{x}_5$, $\boldsymbol{x}_6$ and $\boldsymbol{x}_7$.

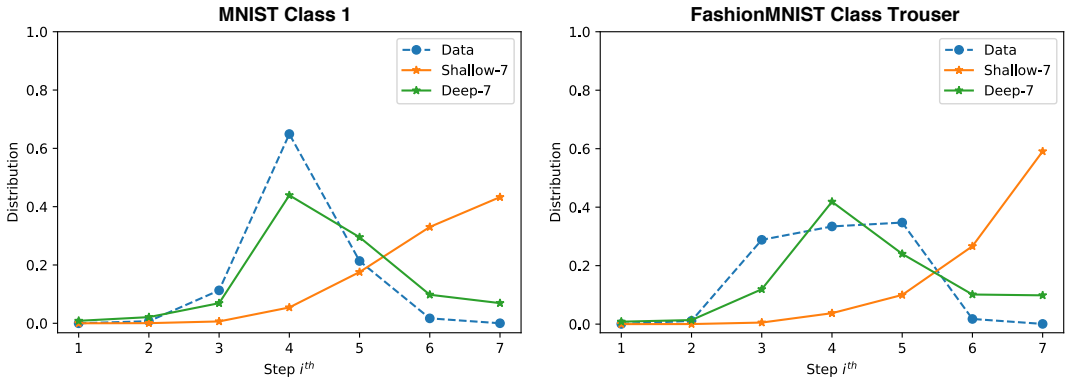


Figure 4.10: Pixel intensity and DTD relevance distribution from Shallow-7 and Deep-7 averaged over MNIST *Class 1* and FashionMNIST *Class Trouser* test population.

4.4.3 Summary

Results from this first experiment seem to suggest that choice of RNN architectures has an impact on quality of its explanation. In particular, as presented in Figure 4.9 and Figure 4.10, quality of deep Taylor decomposition(DTD) explanation is significantly influenced by the architecture. In contrast, we do see such notable effect from sensitivity analysis(SA) and guided backprop(GB) method. In the following experiment, I am going to present a methodical evaluation of this impact in detail.

4.5 Experiment 2 : Majority Sample Classification

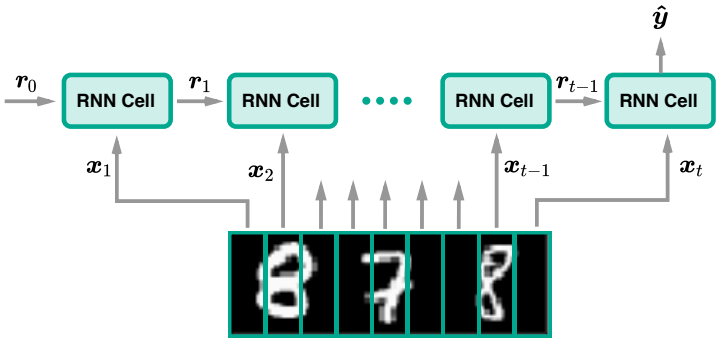


Figure 4.11: Majority Sample Classification Problem

4.5.1 Problem Formulation

When neural network are trained, one can apply explantation techniques to the models to get relevance heatmaps of samples. The heatmap of sample $\boldsymbol{x}$ illustrates important features in x that the trained network utilizes to perform its objective prediction, such as classification. Therefore, one needs to know the ground truth of these latent features in order to methodologically evaluate how well the model distributes relevance scores to $\boldsymbol{x}$'s input space. However, this knowledge is not trivial to find as it is an incident from the optimization process in high-dimensional space that we in fact seek to understand.

To alleviate this challenge, I constructed two artificial datasets using samples from MNIST and FashionMNIST respectively. Let's consider MNIST. The new dataset was constructed as follows: each original sample $\widetilde{\boldsymbol{x}} \in \mathbb{R}^{28,28}$, I randomly selected 2 additional samples : one from the same class of $\widetilde{\boldsymbol{x}}$ and the other one from a different class. Then, these 3 samples are concatenated in random order yielding a sample $\boldsymbol{x} \in \mathbb{R}^{28,84}$ of the new dataset.

With this construction procedure, one possible objective is to train RNNs to predict the majority class in each sequence $(\boldsymbol{x}_t)_{t=1}^{T}$. For example, consider $\boldsymbol{x} = \{8, 7, 8\}$ shown in Figure 4.11, the majority class here is *Class 8*. Because we already know time step t' that are corresponding to original samples $\widetilde{\boldsymbol{x}}$ belonging to the majority group in each sequence $\boldsymbol{x}$, we can simply compute percentage of relevance scores assigned to t' to quantitively evaluate explainability of a RNN architecture.

I also introduces two additional variation of Deep architecture, namely DeepV2 and ConvDeep. DeepV2 has one more layer after the first fully-connected layer, while ConvDeep instead employs a sequence of convolutional and pooling operation after the input layer. Figure X shows details of the architectures.

4.5.2 Result

		Accuracy	
Cell Architecture	**No. variables**	**MNIST**	**FashionMNIST**
Shallow	184330	98.39%	90.15%
Deep	153578	98.42%	90.60%
DeepV2	161386	98.38%	90.43%
ConvDeep	151802	99.07%	92.43%

Table 4.6: Number of trainable variables and model accuracy for Majority Sample Classification experiment.

In this experiment, I chose $T = 12$, hence $(\boldsymbol{x}_t \in \mathbb{R}^{28,7})_{t=1}^{12}$. Table 4.6 shows number of trainable variables and accuracy of the trained models. These trained models have equivalent number of variables and accuracy, hence comparing heatmaps of these models is a fair evaluation.

As can be seen from Figure 4.12, the deeper architecture, the fewer relevant scores distributed to irrelevant region. This effect happens across all explanation methods. This result further supports what has shown in Experiment 1. Moreover, although relevance heatmaps from Shallow-12, Deep-12, and DeepV2-12 generally look noisy, increasing the depth of architecture seems to reduce the noise in the heatmaps. On the other hand, ConvDeep-12 does not only properly assign relevance quantities to the right time steps, but its heatmaps are also sound. In particular, one can easily perceive features of $\boldsymbol{x}$ from ConvDeep-12's GB and DTD heatmaps.

Figure 4.13 presents a quantitive evaluation of the impact from the depth of architecture to the explanation. Here, the measurement is the percentage of relevance quantities assigned to regions of majority class : these regions are indicated by the red triangles in Figure 4.12, averaged over test sample population. Results from Figure 4.13 indicate that the depth of architecture indeed improves quality of the explanations. In particular, the percentage of correct relevance assignment of each explanation technique increases as more layers introduced. This enhancement can be seen clearly from the result of Fash-

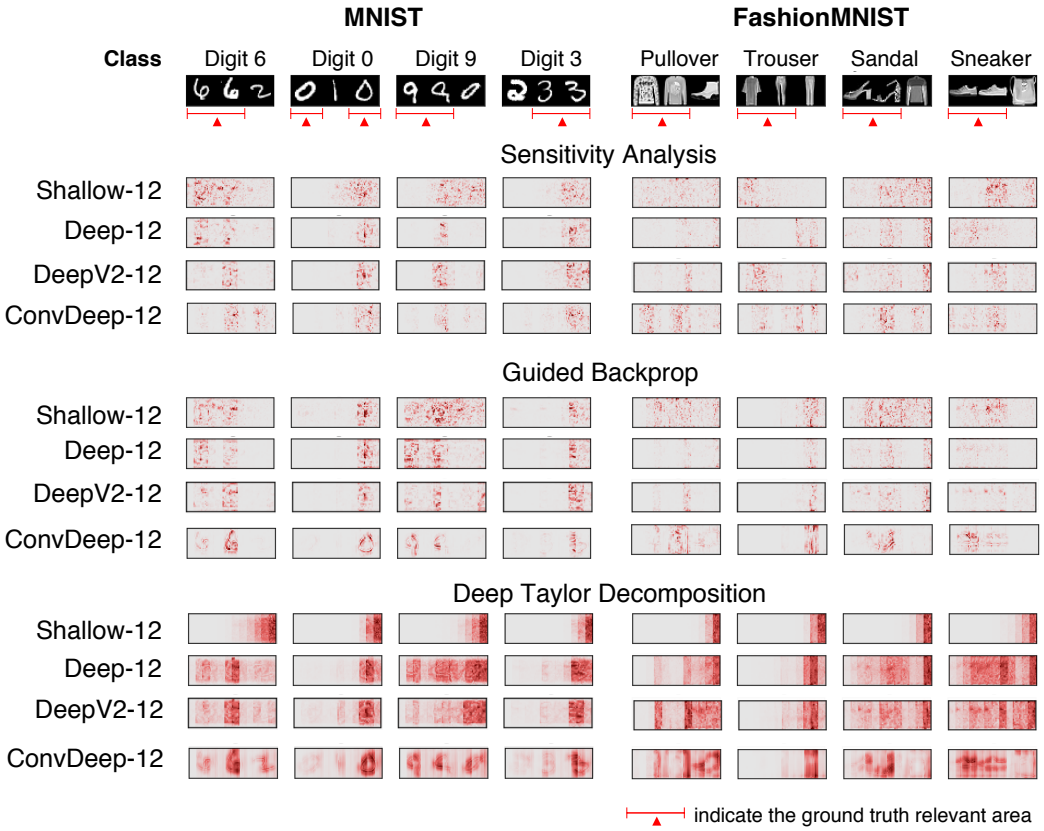


Figure 4.12: Relevance heatmaps for MSC problem

ionMNIST. Moreover, as expected, ConvDeep-12 improves the percentage even more. Additionally, we can observe that the difference between the percentage of the baseline, Shallow-12, and the deep architectures changes with different proposition across methods. In particular, we see the difference of SA and DTD are slightly larger than the difference of GB. This implies that some explanation methods get more benefit from the depth of architecture.

TODO : add cases that convdeep fail

4.5.3 Summary

The outcome of this experiment quantitively confirms that the depth of architecture has impacts on explanation of the model. It also shows that the depth of architecture affects explanation in different level on different methods. More precisely, comparing to guided backprop(GB), quality of sensitivity analysis(SA) and deep Taylor decomposition(DTD) explanation is more depend on the depth.

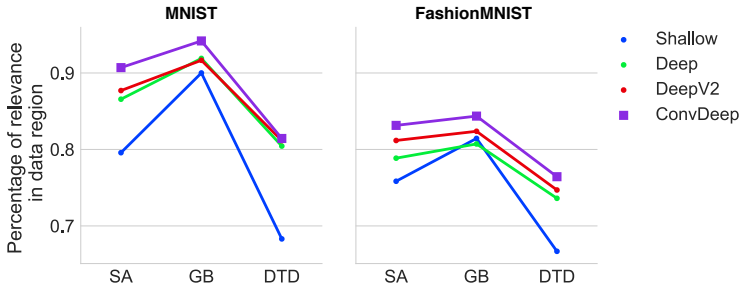


Figure 4.13: Percentage of relevance in data region

4.6 Experiment 3 : Improve Relevance Distribution

The results from previous experiment are a strong evidence that better structured cell architecture leads to better explanation. However, there are some cases that the purposed architectures fail to distribute relevance properly. Figure 4.14 shows such cases. TODO : Here ... should not propagate relevances to ....

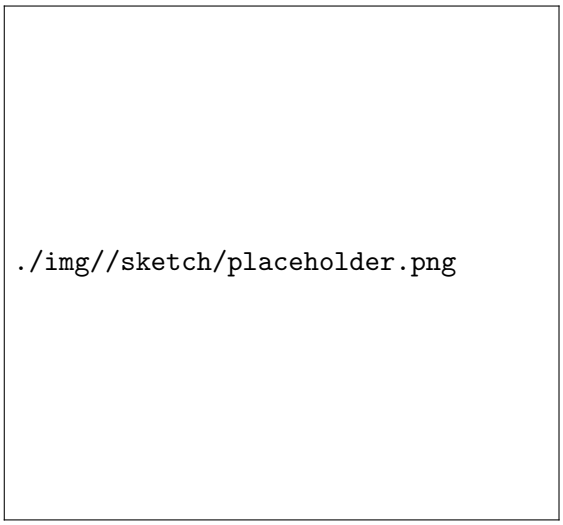


Figure 4.14: Heatmaps of failed cases

Given the motivation above, this experiment aims to extend the proposed architectures further to better address the problem. More precisely, we consider the same classification problem as described in Section 4.5.1 and propose 3 improvements, namely stationary dropout, employing gating units, and literal connections.

4.6.1 Proposal 1 : Stationary Dropout

Dropout is a simple regularization technique that randomly suspends activity of neurons during training. This randomized suspension allows the neurons to learn more specific

representations and reduces chance of overfitting. It is also directly related to explanation quality. Figure 4.15 shows explanations of LeNet trained with different dropout probability.

Figure 4.15: LeNet with various dropout values

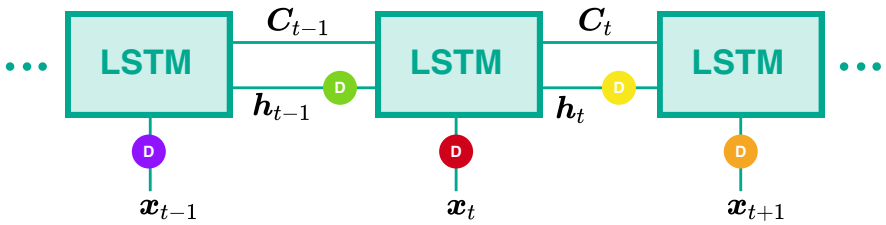
However, unlike typical feedforward architectures, RNN layers are reused across time step, hence a question arises whether the same neurons in those layers should be suspended or they should be different neurons. Figure 4.16a and Figure 4.16b illustrates these 2 different approaches where different colors represent different dropping activities. In particular, this stationary dropout was first proposed by [Gal and Ghahramani, 2016] who applied the technique to LSTM and GRU and found improvements on language modeling tasks.

4.6.2 Proposal 2 : Gating units

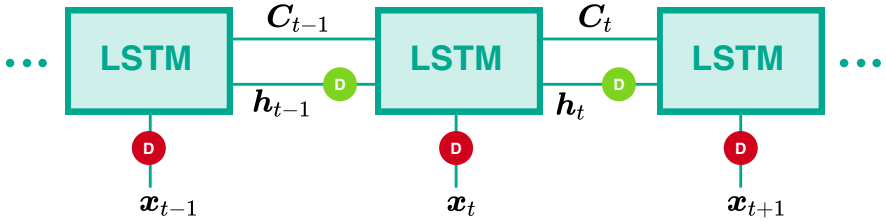
It is already shown that gating units and addictive updates are critical mechanisms that enable LSTM to learn long term dependencies [Greff et al., 2017, Jozefowicz et al., 2015]. However, LSTM is not readily applicable for methods we are considering in this thesis. More precisely, the use of sigmoid and tanh activations violates the assumption of GB and DTD. Therefore, we propose a slight modified version of LSTM where ReLU activations are used to compute cell state candidates $\widetilde{C}_t$ instead of tanh functions. This results $C_t \in \mathbb{R}^+$, hence the tanh activation for h_t is also removed. From the following, we will refer this architecture as R-LSTM to differentiate from the original. Figure 4.16 presents an overview of R-LSTM architecture.

4.6.3 Proposal 3 : Convolutional layer with literal connections

As discussed in Section 3.1.3, convolution and pooling operator enable NNs to learn hierarchical representations, which are directly beneficial to explanation quality. The



(a) Naive Dropout



(b) Stationary Dropout

ConvDeep architecture we proposed in Section does not seem to exploit this properly because it has recurrent connections only layers after the convolutional and pooling layers. This can be analogically viewed that ConvDeep shares high-level features between step instead of low-level features. This might lead to obscure low-level features in the explanation. Figure 4.17 shows the architecture with literal connections are highlighted in red.

Therefore, we propose an extension of ConvDeep architecture where result of convolution operator is also incorporated into the operator in the next step. We name this connection as *literal connection* and refer ConvDeep$^+$ to the proposed architecture.

4.6.4 Setting

In this experiment, we divided the experiment into TODO : 3? parts. The first part focuses on stationary dropout and gating units proposal. The Deep architecture is used as the baseline. We also added one layer with 256 neurons between input and 75 R-LSTM cells to make it comparable to the Deep architecture.

The literal connection proposal is considered in the second part of the experiment where we compare the result to the Deep architecture. The last part simply combines promising improvements together. Setting of hyperparameters are the same as in Section 4.3.

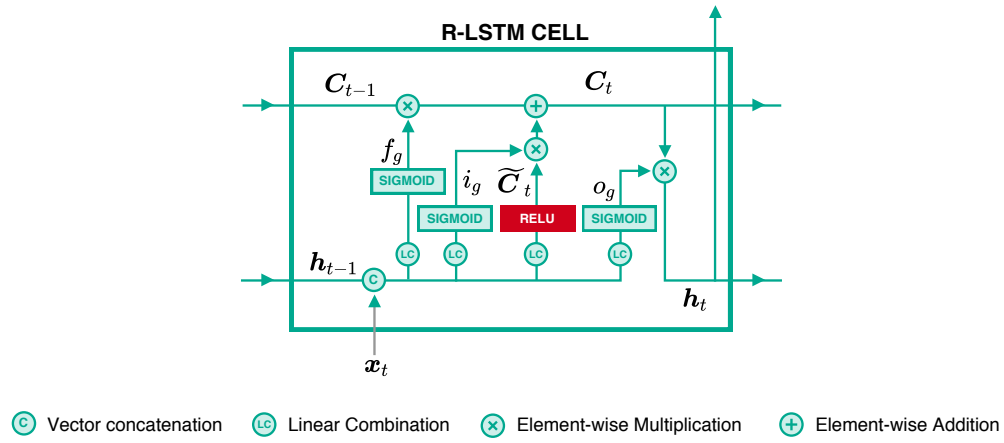


Figure 4.16: R-LSTM Structure

4.6.5 Result

Table shows accuracy and number of parameters of Deep and R-LSTM architecture.

4.6.6 Summary

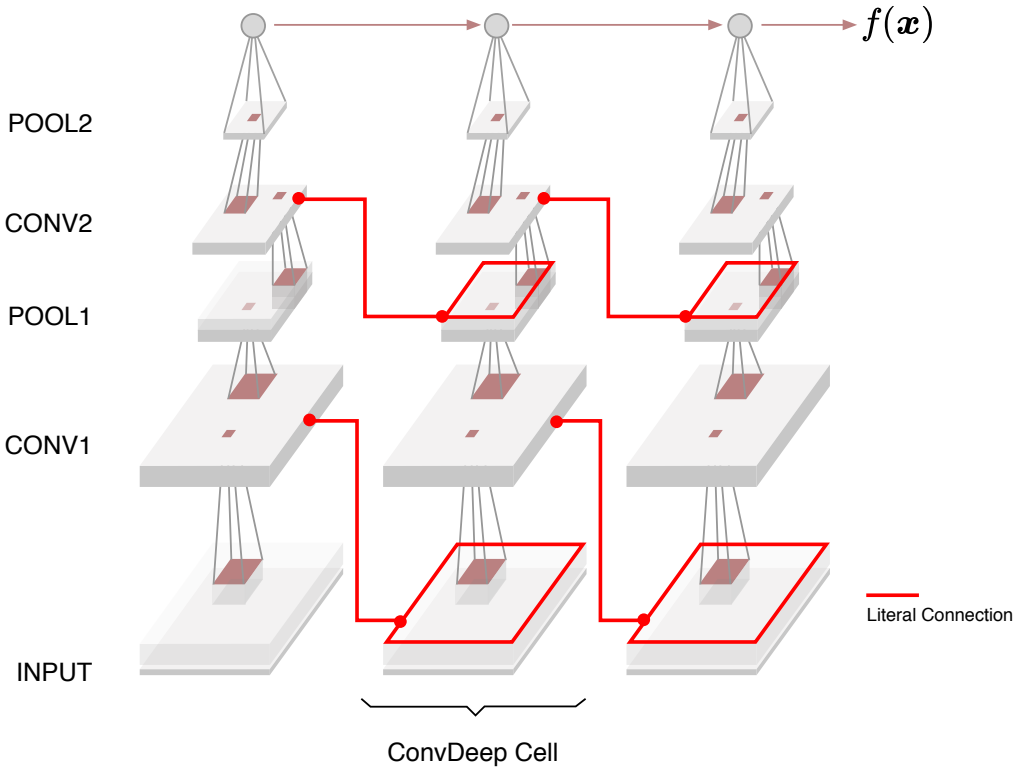


Figure 4.17: ConvDeep with literal connections(ConvDeep$^+$)

References

[Abadi et al., 2016] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mane, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viegas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2016). TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems.

[Bach et al., 2016] Bach, S., Binder, A., Montavon, G., Muller, K.-R., and Samek, W. (2016). Analyzing classifiers: Fisher vectors and deep neural networks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 2912–2920.

[Binder et al., 0906] Binder, A., Montavon, G., Lapuschkin, S., Müller, K.-R., and Samek, W. (2016/09/06). Layer-Wise Relevance Propagation for Neural Networks with Local Renormalization Layers. In Artificial Neural Networks and Machine Learning – ICANN 2016, Lecture Notes in Computer Science, pages 63–71. Springer, Cham.

[Cho et al., 2014] Cho, K., van Merriënboer, B., Gülçehre, Ç., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), pages 1724–1734. Association for Computational Linguistics.

[Erhan et al., 2010] Erhan, D., Courville, A., and Bengio, Y. (October 2010). Understanding Representations Learned in Deep Architectures.

[Gal and Ghahramani, 2016] Gal, Y. and Ghahramani, Z. (2016). A Theoretically Grounded Application of Dropout in Recurrent Neural Networks. In Lee, D. D., Sugiyama, M., Luxburg, U. V., Guyon, I., and Garnett, R., editors, Advances in Neural Information Processing Systems 29, pages 1019–1027. Curran Associates, Inc.

[Greff et al., 2017] Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R., and Schmidhuber, J. (2017). LSTM: A search space odyssey. IEEE transactions on neural networks and learning systems.

[He et al., 2015] He, K., Zhang, X., Ren, S., and Sun, J. (2015). Deep Residual Learning for Image Recognition. CoRR, abs/1512.03385.

[Hochreiter and Schmidhuber, 1997] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. Neural computation, 9(8):1735–1780.

[Jia et al., 2014] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. (2014). Caffe: Convolutional Architecture for Fast Feature Embedding.

[Jozefowicz et al., 2015] Jozefowicz, R., Zaremba, W., and Sutskever, I. (2015). An empirical exploration of recurrent network architectures. Journal of Machine Learning Research.

[Kingma and Ba, 2014] Kingma, D. P. and Ba, J. (2014). Adam: A Method for Stochastic Optimization. In Proceedings of the 3rd International Conference on Learning Representations (ICLR).

[Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q., editors, Advances in Neural Information Processing Systems 25, pages 1097–1105. Curran Associates, Inc.

[LeCun et al., 2001] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (2001). Gradient-Based Learning Applied to Document Recognition. In Intelligent Signal Processing, pages 306–351. IEEE Press.

[LeCun and Cortes, 2010] LeCun, Y. and Cortes, C. (2010). MNIST handwritten digit database.

[Montavon et al., 2017a] Montavon, G., Lapuschkin, S., Binder, A., Samek, W., and Müller, K.-R. (May 1, 2017a). Explaining nonlinear classification decisions with deep Taylor decomposition. Pattern Recognition, 65:211–222.

[Montavon et al., 2017b] Montavon, G., Samek, W., and Müller, K.-R. (2017b). Methods for Interpreting and Understanding Deep Neural Networks.

[Pascanu et al., 2012] Pascanu, R., Mikolov, T., and Bengio, Y. (2012). Understanding the exploding gradient problem. CoRR, abs/1211.5063.

[Simonyan et al., 2013] Simonyan, K., Vedaldi, A., and Zisserman, A. (2013). Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps. CoRR, abs/1312.6034.

[Simonyan and Zisserman, 2014] Simonyan, K. and Zisserman, A. (2014). Very Deep Convolutional Networks for Large-Scale Image Recognition. CoRR, abs/1409.1556.

[Springenberg et al., 2014] Springenberg, J. T., Dosovitskiy, A., Brox, T., and Riedmiller, M. A. (2014). Striving for Simplicity: The All Convolutional Net. CoRR, abs/1412.6806.

[Szegedy et al., 2014] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2014). Going Deeper with Convolutions.

[Tieleman and Hinton, 2012] Tieleman, T. and Hinton, G. (2012). Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude.

[Xiao et al., 2017] Xiao, H., Rasul, K., and Vollgraf, R. (2017). Fashion-MNIST: A Novel Image Dataset for Benchmarking Machine Learning Algorithms.

[Zagoruyko and Komodakis, 2016] Zagoruyko, S. and Komodakis, N. (2016). Wide Residual Networks.

[Zeiler, 2012] Zeiler, M. D. (2012). ADADELTA: An Adaptive Learning Rate Method. CoRR, abs/1212.5701.

[Zeiler and Fergus, 2013] Zeiler, M. D. and Fergus, R. (2013). Visualizing and Understanding Convolutional Networks. CoRR, abs/1311.2901.