Department of Informatics, University of Zürich

MSc Basic Module

Implementing Deconvolution to Visualize and Understand Convolutional Neural Networks

Xiaozhe Yao

Matrikelnummer: 19-759-570

Email: xiaozhe.yao@uzh.ch

August 15, 2020

University of Zurich[UZH]

Department of Informatics

1 Introduction

Deep convolutional neural networks (CNNs) have played an important role in computer vision tasks, such as image classification, object detection, etc. However, as a black-box model, it is still unclear how these neural networks work. Hence, many researchers are working towards the interpretability of deep convolutional neural networks, such as [MF14]. They use the deconvolution network to visualize the features in convolutional layers. In their approach, there are three important components: 1) the pretrained deep convolutional neural network, including convolutional layers, pooling layers, fully connected layers and ReLu layers. 2) the deconvolution operation and 3) the max-unpooling operation. The goal of this basic module is to reimplement these components from scratch and apply them to visualize the intermediate activations in convolution layers. Having these implementations, we will also compare our results with a PyTorch implementation.

This report will be organized into the following sections:

- **Preliminaries**. In this section, we will introduce the preliminary knowledge that we need. More specifically, we will introduce the *Convolutional Neural Network*, *Deconvolution Operation*, *Stochastic Gradient Descent and Backpropagation*, *Differentiation of Linear Transformations* and *Numerical Differentiation*. Among them, the differentiation laws of a linear transformation are the most important and fundamental knowledge of the whole report.

- **Approach**. In this section, we will thoroughly explain the mechanisms of essential components in our system: *the fully connected layer*, *convolutional layer*, *deconvolution layer*, *max-pooling* and *max-unpooling*. Besides, we will also induce the computational rules for those layers.

- **Running Example**. In this section, we will present a running example of using our approach to perform image classification and visualization of the intermediate activations.

- **Implementation**. In this section, we will explain and demonstrate how we implement the training, evaluating and interpreting process. More specifically, we will present the implementation of backpropagation, fully connected layer and some other utilities.

- **Experiment**. In this section, we will present the architecture and training settings we used in the classification model. With the classification model, we will give the classification results on a given dataset, and try to interpret the intermediate activations with a deconvolutional network. Besides these, we will also compare the results of using our naive implementation and a PyTorch implementation.

2 Preliminaries

Convolutional Neural Network: Convolutional Neural Network (CNN) is a class of neural networks, and has been proven to be effective for most computer vision tasks. In a CNN architecture for image classification, there are usually three important components: the convolutional layer, the pooling layer and the fully connected layer. The first two types are designed to extract high-level features from images, and the fully connected layer can be used as a classifier to output the classification results. In a convolutional neural network, the convolutional and fully connected layers are equipped with parameters called *weight* and *bias*, which will be updated during training. In this work, we implemented these components along with other necessary components, such as activations (ReLu function), losses (Cross-Entropy Loss), etc.

Deconvolution: The two fundamental components of the CNN are convolutional layers and pooling layers, which works together to transform images into feature maps. Deconvolutional operation is a transformation that goes in the opposite direction of a normal convolution operation, i.e. from the feature maps that we extracted with convolution operation to something that has the shape of the input to it. After being introduced, deconvolution has been used in many fields such as pixel-wise segmentation, generative models, etc. In this work, we use deconvolution to map the intermediate feature maps back to the input pixel space. With this approach, we can show the relationship between the input patterns and the activations in the feature maps. There are also two components in the approach, called deconvolutional layers and unpooling layers, and we will explain these two concepts in more detail in *Section 3 Approach*.

Stochastic Gradient Descent: In neural networks, we want to find a function $\hat{y} = F(x)$ such that $y - F(x)$ is minimal. The function that we are looking for is usually non-linear, non-convex and there is generally no formula for it. As a result, gradient descent becomes one of the most popular methods to find the local minimum of the function. The method is based on a fact that the function f decreases fastest along the direction of the negative gradient. Formally, we can define a function that measures the difference between $\hat{y}$ and y, for example $f = y - \hat{y}$ and assume that a is the only parameter in f. Then if we let $a_{n+1} = a_n - \epsilon \nabla_a f$ and we want to find the lowest value of $f(a)$ around the point a. Then if ϵ is small enough, we will have $f(a_{n+1}) \leq f(a_n)$ and $f(a_{n+1})$ is the smallest value around a small enough interval of a. Considering this, if we want to find the local minimal of the function f, we can start at a random point a_0, and follows the negative direction of the gradient. With this approach, we will have a sequence $a_1, a_2, \cdots a_n$ that satisfies $a_{n+1} = a_n - \epsilon \nabla_a f$. Then the output of the function f will satisfy the rule that $f(a_n) \leq f(a_{n-1}) \leq \cdots \leq f(a_0)$. By doing so, we could find an approximate value a_n such that $f(a_n)$ is the local minimal.

Backpropagation: In the process of gradient descent, we found that we need to compute the gradient of our function f in every step. Backpropagation, as an application of the chain rule, is an efficient algorithm for calculating the gradient in deep neural networks. In short, it first computes the gradient of the loss function to the weight of the last layer in a neural network, and passes the gradient of the loss function to the input of the layer to previous layers. There are two bases for the algorithm:

- In the ith layer, we can receive the gradient of the loss ℓ with respects to the output of ith layer, i.e. $\frac{\partial \ell}{\partial \hat{y}^{(i)}}$ is known to us.
- Since the output of $(i-1)$th layer is the input of the ith layer, we have $\frac{\partial \ell}{\partial x^{(i)}} = \frac{\partial \ell}{\partial \hat{y}^{i-1}}$

Having these two bases, we could compute the gradient of the loss ℓ with respect to the weight and input of every layer by applying chain rules. For example, $\frac{\partial \ell}{\partial w^{(i)}} = \frac{\partial \ell}{\partial \hat{y}^{(i)}} \frac{\partial \hat{y}^{(i)}}{\partial w^{(i)}}$ where we only need to know how to compute $\hat{y}^{(i)}$ with $w^{(i)}$. With these, we could efficiently compute the loss value with respect to every parameter in the neural network and update them with the SGD method.

Notations In the below sections, we will by default use the following notations:

- $x_j^{(i)}$ is the jth input of the ith layer. In fully connected layers, all the inputs form a vector, thus we can use $x_j^{(i)}$ to denote them. However, in convolutional layers, the inputs form a 2d matrix, and in that case, we will use $x_{kj}^{(i)}$ to denote the individual input. The formed vectors (in fully connected layers) or matrices (in convolutional layers) will be represented by $X^{(i)}$.
- $\hat{y}_j^{(i)}$ is jth the output of the ith layer. If the ith layer is the last layer, there might be some corresponding ground truths. In this case, the ground truths will be denoted as $y_j^{(i)}$. We will use $\hat{Y}^i$ to denote the vector or matrix that contains all the outputs of the ith layer. Correspondingly, the vector or matrix of the ground truth will be denoted by $Y^{(i)}$
- ℓ is the value of loss function. In our classification model, we use the cross-entropy loss and it is defined as $\ell = -\sum_i^n y_j^{(i)} log(\hat{y}_j^{(i)})$ where i is the index of the last layer. In regression models, it can be defined as $\ell = \frac{1}{n}\sum_{i=1}^n (\hat{y}_i - y_i)^2$ (mean square loss). In the below examples, we will by default use the mean square loss as it is easier to compute.
- $\nabla_X f$ refers to the derivative of function f with respect to X, i.e. $\nabla_X f = \frac{\partial f}{\partial X}$. In the following sections, we will use $\nabla^{(i)}$ to represent the gradient of the loss value with respect to the output of the ith layer. Formally, we have $\nabla^{(i)}$=$\frac{\partial \ell}{\partial \hat{Y}^{(i)}}$. Since the output of the ith layer is also the input of the $(i+1)$th layer, we also have $\nabla^{(i)} = \frac{\partial \ell}{\partial X^{(i+1)}}$. In case $\nabla^{(i)}$ is a matrix, we will use $\nabla_{kj}^{(i)}$ to denote the (k, j) element in the matrix as well.
- There are some other notations we need in the following sections: $\mathbb{R}$ is for the set of real numbers, $\mathbb{R}^{m \times n}$ is for an $m \times n$ real matrix and ϵ is a small enough real number.

Differentiation of Linear Transformation of Matrices As suggested above, we will need to compute the derivatives of the loss functions to the parameters in neural networks. As we usually represent the data (such as the input, output, other parameters, etc) in neural networks as matrices, and the most fundamental transformations in neural networks are linear, it is essential to understand how to compute the derivatives of a linear transformation of matrices by using the chain rule.

Since there is no such concept "Layer" in this process, we will use X and Y to denote the matrices and x_{ij} and y_{kl} to represent entries in matrices without the superscripts.

Assume that we have $f(Y) : \mathbb{R}^{m\times n} \to \mathbb{R}$ and a linear tranformation $g(X) : \mathbb{R}^{p\times n} \to \mathbb{R}^{m\times n}, Y = g(X) = AX + B$, where $A \in \mathbb{R}^{m\times p}, B \in \mathbb{R}^{m\times n}$. We can compute the derivatives of f with respect to X as the following:

- We know, at the point x, if there are two intermediate variables $u = \phi(x)$ and $v = \psi(x)$ that have partial derivatives with respect to x defined, then the composited function $f(u, v)$ has partial derivatives with respect to x defined and can be computed as $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial u}\frac{\partial u}{\partial x} + \frac{\partial f}{\partial v}\frac{\partial v}{\partial x}$. In our case, there might be several intermediate variables y_{kl}, hence we have $\frac{\partial f}{\partial x_{ij}} = \sum_{kl} \frac{\partial f}{\partial y_{kl}}\frac{\partial y_{kl}}{\partial x_{ij}}$ (1).

- Let a_{ij} and b_{ij} represent elements in A and B, then $y_{kl} = \sum_s a_{ks}x_{sl} + b_{kl}$. Hence we have $\frac{\partial y_{kl}}{\partial x_{ij}} = \frac{\partial \sum_s a_{ks}x_{sl}}{\partial x_{ij}} = \frac{\partial a_{ki}x_{il}}{\partial x_{ij}} = a_{ki}\delta_{lj}$ (2). Here δ_{lj} is defined as $\delta_{lj} = 1$ when $l = j$, otherwise $\delta_{lj} = 0$. Intuitively, we know that for the single pair (x_{ij}, y_{kl}), there is a relation $y_{kl} = a_{ki}x_{il} + b_{kl}$. Hence the derivative of y_{kl} with respect to x_{ij} is a_{ki} only when $l = j$, otherwise, the derivative will be 0.

- Take (2) into (1), we will have $\frac{\partial f}{\partial x_{ij}} = \sum_{kl} \frac{\partial f}{\partial y_{kl}}\frac{\partial y_{kl}}{\partial x_{ij}} = \sum_{kl} \frac{\partial f}{\partial y_{kl}}a_{ki}\delta_{lj} = \sum_k \frac{\partial f}{\partial y_{kj}}a_{ki}$ (because only y_{kj} will be kept). In this equation, we know that a_{ki} is the ith row of A^T and $\frac{\partial f}{\partial y_{kj}}$ is the (k, j) element in the gradient of f with respect to Y. In summary, this equation tells us that the derivative of f with respect to x_{ij} is the dot product of the ith row of A^T and the jth column of $\nabla_Y f$.

- Now that we have already known that $\frac{\partial f}{\partial x_{ij}}$ is the dot product of the ith row of A^T and the jth column of $\nabla_Y f$. Then for $\frac{\partial f}{\partial X}$, we have

$$\frac{\partial f}{\partial X} = \begin{bmatrix} \frac{\partial f}{\partial x_{11}} & \cdots & \frac{\partial f}{\partial x_{1n}} \\ \vdots & \frac{\partial f}{\partial x_{ij}} & \vdots \\ \frac{\partial f}{\partial x_{p1}} & \cdots & \frac{\partial f}{\partial x_{pn}} \end{bmatrix} = A^T\nabla_Y f$$

  because every element in $\frac{\partial f}{\partial X}$ equals to a inner product of a row and a column.

- It is also common that $g(X)$ is defined as $g(X) : \mathbb{R}^{m\times p} \to \mathbb{R}^{m\times n}, Y = g(X) = XC + D$ where $C \in \mathbb{R}^{p\times n}$ and $D \in \mathbb{R}^{m\times n}$. In this case, we have $f(Y) : \mathbb{R}^{m\times n} \to \mathbb{R}$ defined as well. Then to compute $\frac{\partial f}{\partial X}$, we first consider Y^T and we have $Y^T = (XC + D)^T = C^T X^T + D^T$. Then by the laws we have found above, we immediately know

that $\nabla_{X^T} f = (C^T)^T \nabla_{Y^T} f = C \nabla_{Y^T} f$. Therefore, we have $\nabla_X f = (\nabla_{X^T} f)^T = (C \nabla_{Y^T} f)^T = (\nabla_{Y^T} f)^T C^T = (\nabla_Y f) C^T$

In summary, if we have two functions, $f(Y)$ takes a matrix and returns a scalar, and a linear tranformation function $g(X)$, then we can perform the differentiation using the chain rule. More specifically, we have found two laws:

- If the linear transformation is defined as $g(X) = AX + B$ (we call this left multiplcation), then we have $\nabla_X f = A^T \nabla_Y f$. (Law 1)
- If the linear transformation is defined as $g(X) = XC + D$ (we call this right multiplcation), then we have $\nabla_X f = (\nabla_Y f) C^T$. (Law 2)

Note: We should be careful about the independent variable. Here we are computing the gradient of the function f with respect to X, we can also compute the gradient of the function f with respect to C. In that case, we should use a different law.

These two laws are the most important and fundamental conclusions we have in the whole work, and we will find out that the essential components of a convolutional neural network (fully connected, convolution, etc) are linear transformations and the loss function is the $f(Y)$ that we have here. As a result, we will show how to transform those components into a linear transformation and these transformations will form the mainline of the *Section 3 Approach*.

Numerical Differentiation Besides manually working out the derivatives, we can also estimate the derivatives with numerical approximation. numerical differentiation is an algorithm for estimating the derivative of a mathematical function using the values of the function.

The simplest method, also known as Newton's differentiation quotient is by using the finite difference approximations. More specifically, if we have a function $f(x)$ and we want to compute the derivative of f, we can approximate it by computing the slope of a nearby secant line through the points $(x, f(x))$ and $(x + \epsilon, f(x + \epsilon))$. The slope of this secant line will be $\frac{f(x+\epsilon)-f(x)}{\epsilon}$, and the derivative is the tangent line at the point x. As ϵ approaches 0, the slope of the secant line approaches the slope of the tangent line. Therefore, the true derivative of f at the point x can be defined as $f'(x) = \lim_{\epsilon \to 0} \frac{f(x+\epsilon)-f(x)}{\epsilon}$. Then by this nature, we could manually choose a small enough ϵ, and to approximately approach the tangent line, i.e. the derivative of the function f at x.

As we know the point $x + \epsilon$ is at the right of x, the form $\frac{f(x+\epsilon)-f(x)}{\epsilon}$ is called right-sided form. Besides this form, we can also approach the tangent line from the left side and right side (the two-sided form) at the same time. To do so, we compute the slope of a nearby secant line through the points $(x - \epsilon, f(x - \epsilon))$ and $(x + \epsilon, f(x + \epsilon))$ by $\frac{f(x+\epsilon)-f(x-\epsilon)}{2\epsilon}$. This form is a more accurate approximation to the tangent line than the one-sided form and therefore we will use the two-sided form in the following sections.

In order to verify that we are working out the derivatives correctly, we will involve the numerical differentiation as a way of cross-validation, and increase our confidence in the correctness of induction. In *3.1.1 Fully Connected*, we will show how to perform the numerical differentiation in a concrete and simple example.

3 Approach

In this work, we will build a classification model to categorize the input image, and an interpretation model that performs the inverse operation of the classification model and try to interpret the intermediate activities. For the classifier, we use a VGG-like network consisted of convolutional layers, max-pooling layers and fully connected layers, while for the interpretation model, we use the corresponding deconvolution layers, max-unpooling layers to perform the inverse operations. In this section, we will thoroughly induce the computation rules of the forward and backward pass of these layers.

3.1 Classification Model

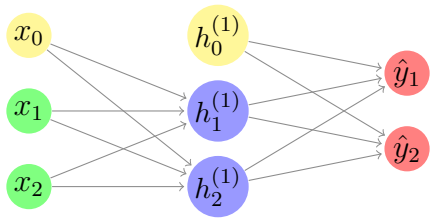
After proposed, VGG-net [SZ14] has been proven to be effective for image classification task and hence we applied a VGG-like architecture in our neural network. There are three essential components inside a VGG-like network: the fully connected layer, the convolutional layer and the pooling layer. In order to fit the classifier to our dataset with the SGD method, we need to know how to compute the forward and backward pass of these layers. As introduced in *Section 2 Preliminaries*, we will try to convert those layers into linear transformations of matrices, and reuse the two laws that we have found when computing the gradient.

3.1.1 Fully Connected

Definition The fully connected layer is the simplest and most fundamental neural network component as it connects all the nodes in a layer to all the nodes in the next layer. For example, in Fig. 3.1.1 we present a simple fully connected layer that connects all nodes, in which yellow nodes represent the bias (the nodes that are independent of the output of the previous layer) in each layer, green nodes represent the input to the neural network, blue nodes represent the intermediate activities and red nodes represent the output results.

Input	Hidden	Output
layer 0	layer 1	layer 2

Fig 3.1.1 Illustration of the fully connected layers.



From the definition of fully connected layer, we know that in the ith layer, if there are $n^{(i)}$ input nodes (in our figure, $n^{(1)} = 2$ (the green nodes), $n^{(2)} = 2$ (the blue nodes)) and $m^{(i)}$ output nodes (in our figure, $m^{(0)} = 2$ (the blue nodes), $m^{(1)} = 2$ (the red nodes)), there will be $m^{(i)} \times n^{(i)}$ relations between them. These relations can be represented as a $m^{(i)} \times n^{(i)}$ matrix $w^{(i)}$ (weight). Besides of these relations, there will be $m^{(i)}$ biases that can be represented as a $m^{(i)}$d vector $b^{(i)}$ (bias). (in our figure, $b^{(0)} = 2$ (x_0 to $h_1^{(1)}$ and x_0 to $h_2^{(1)}$), $b^{(1)} = 2$ ($h_0^{(1)}$ to $\hat{y}_1$ and $h_0^{(1)}$ to $\hat{y}_2$))

Forward Pass With the above notations, we can directly compute the forward pass. For example, in Fig. 3.1.1, we can compute $h_1^{(1)}$ as $h_1^{(1)} = x_1 w_{11}^{(1)} + x_2 w_{21}^{(1)} + x_0$ where $w_{jk}^{(i)}$ is the relation between x_j and $h_k^{(i)}$. In matrix form, we have $\hat{Y} = Xw + b$ where w is the weight matrix, X is the input vector and b is the bias vector. In the matrix form, w is a $n \times m$ matrix, X is a nd vector and b is a md vector.

Backward Pass In the training process, we want to update the relations between the input nodes and the output nodes. More specifically, we want to know how the weight w and bias b will impact the loss value, i.e. we want to compute $\frac{\partial \ell}{\partial w}$ and $\frac{\partial \ell}{\partial b}$. In the backward pass, the ith layer will receive the gradient of the loss value with respect to the input of the $(i+1)$th layer. As the input of the $(i+1)$th layer is the output of the ith layer, we have $\frac{\partial \ell}{\partial \hat{y}_i}$ known indeed.

With these conditions, we can apply the laws we have to compute the gradients. We will have

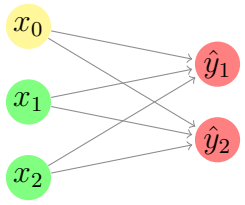
- $\frac{\partial \ell}{\partial w} = X^T \nabla^{(i)}$ (Using the Law 1)
- $\frac{\partial \ell}{\partial b} = \frac{\partial \ell}{\partial \hat{y}^i} \frac{\partial \hat{y}^i}{\partial b} = \nabla^{(i)}$

With these results, we can update the weight and bias in the ith layer with $w^{new} = w^{old} - \epsilon \nabla^{(i)} x^T$ and $b^{new} = b^{old} - \epsilon \nabla^{(i)}$ where ϵ is a preset hyperparameter called learning rate. After updating these parameters, we need to pass the gradient of the loss with respect to the input of this layer to the previous layer. Formally, we also have $\frac{\partial \ell}{\partial x^i} = \nabla^{(i)} w^T$ (Using the Law 2).

Example Assume we have a simple fully connected neural network with only two layers, and each layer have two inputs and two outputs. It can be visualized as in Fig. 3.1.2:

Input layer | Output layer

Fig 3.1.2 Illustration of the fully connected layers.



In the above example, we initialize the values as the following:

- In the input layer, we assume the inputs are $x_1 = 0.1$ and $x_2 = 0.5$. Thus the input can be represented by a 2d vector $X = [0.1, 0.5]$
- Since there are 2 input nodes, and 2 output nodes, the weight can be represented as a 2×2 matrix. We assume it to be $w^{(0)} = \begin{bmatrix} w_{11}^{(0)} & w_{12}^{(0)} \\ w_{21}^{(0)} & w_{22}^{(0)} \end{bmatrix} = \begin{bmatrix} 0.15 & 0.20 \\ 0.25 & 0.30 \end{bmatrix}$ where $w_{kj}^{(i)}$ is the weight between $x_k^{(i)}$ and $y_j^{(i)}$. For example, $w_{11}^{(0)}$ is the weight between x_1 and y_1 in our case.
- Since there are 2 output nodes, the biases can be represented as a 2d vector. In our example, we assume it to be $b^{(0)} = [0.35, 0.45]$.
- In order to compute the backward pass, we need to give a ground truth of the output nodes, so that we can compute the loss value and perform the backpropagation. Here we assume that the ground truth is $y_1 = 0.01$ and $y_2 = 0.99$. Thus the output can be represented by a 2d vector $Y = [0.01, 0.99]$.

Forward Pass In the forward pass, we want to compute $\hat{Y}$. In our case, $\hat{Y} = Xw^{(0)} + b^{(0)} = [0.1, 0.5] \begin{bmatrix} 0.15 & 0.20 \\ 0.25 & 0.30 \end{bmatrix} + [0.35, 0.45] = [0.49, 0.62]$

Then with the predicted Y, we can compute the loss value as $\ell = \frac{1}{2}[(0.49 - 0.01)^2 + (0.62 - 0.99)^2] = 0.18365$. We also need to compute the gradient of the loss with respect to the output as $\frac{\partial \ell}{\partial \hat{y}_1} = \hat{y}_1 - y_1 = 0.48$, $\frac{\partial \ell}{\partial \hat{y}_2} = \hat{y}_2 - y_2 = -0.37$. Hence, we have $\nabla^{(1)} = [0.48, -0.37]$.

Backward Pass Then in the backward pass, as shown above, we can compute the gradient of the loss with respect to the weight, bias and the input. In our case, we have

- $\frac{\partial \ell}{\partial w} = X^T \nabla^{(1)} = [0.1, 0.5]^T [0.48, -0.49] = \begin{bmatrix} 0.048 & -0.037 \\ 0.24 & -0.185 \end{bmatrix}$

- $\frac{\partial \ell}{\partial b} = \nabla^{(1)} = [0.48, -0.37]$
- $\frac{\partial \ell}{\partial x} = \nabla^{(1)} w^T = [0.48, -0.37] \begin{bmatrix} 0.15 & 0.20 \\ 0.25 & 0.30 \end{bmatrix} = [-0.002, 0.009]$

Numerical Verification In order to verify that our derivatives are induced correctly, we can compute the derivatives numerically. In order to do so, we firstly let $\epsilon = 0.01$ and then we can compute the derivatives of the loss value with respect to the w as the following:

- For $w_{11}^{(0)}$, we first let $w_{left} = w_{11}^{(0)} - \epsilon = 0.14$ and $w_{right} = w_{11}^{(0)} + \epsilon = 0.16$. Then we have
  - $y_{left} = Xw_{left} + b^{(0)} = [0.1, 0.5] \begin{bmatrix} 0.14 & 0.20 \\ 0.25 & 0.30 \end{bmatrix} + [0.35, 0.45] = [0.489, 0.62]$. At this time, we can compute the loss value as $\ell_{left} = \frac{1}{2}[(0.489 - 0.01)^2 + (0.62 - 0.99)^2] = 0.1831705$.
  - $y_{right} = Xw_{left} + b^{(0)} = [0.1, 0.5] \begin{bmatrix} 0.16 & 0.20 \\ 0.25 & 0.30 \end{bmatrix} + [0.35, 0.45] = [0.491, 0.62]$. At this time, we can compute the loss value as $\ell_{left} = \frac{1}{2}[(0.491 - 0.01)^2 + (0.62 - 0.99)^2] = 0.1841305$.
  - Then we can compute the derivative of the loss value with respect to w_{11} numerically as $\frac{\partial \ell}{\partial w_{11}^0} = \frac{\ell_{right} - \ell_{left}}{2\epsilon} = \frac{0.1841305 - 0.1831705}{0.02} = 0.048$.
- Similarly, we can do the same procedures for $w_{12}^{(0)}$, $w_{21}^{(0)}$ and w_{22}^0. After computation, we have $\frac{\partial \ell}{w_{12}^{(0)}} = \frac{0.1832805 - 0.1840205}{0.02} = -0.037$, $\frac{\partial \ell}{w_{21}^{(0)}} = \frac{0.1860625 - 0.1812625}{0.02} = 0.24$ and $\frac{\partial \ell}{w_{22}^{(0)}} = \frac{0.184225 - 0.187925}{0.02} = -0.185$. With these four values we can form the matrix of the loss value with respect to the weight matrix as $\frac{\partial \ell}{\partial w} = \begin{bmatrix} 0.048 & -0.037 \\ 0.24 & -0.185 \end{bmatrix}$ and this result is consistent with the matrix we get from our inductions. By far we verified that we are computing the derivatives correctly for the weight.

Similarly, we proceed to verify the biases as following:

- There are two biases in our case, $b_1^{(0)} = 0.35$ and $b_2^{(0)} = 0.45$. For $b_1^{(0)}$, we define $b_{left} = b_1^{(0)} - \epsilon = 0.34$ and $b_{right} = b_1^{(0)} + \epsilon = 0.36$. Then we have
  - $y_{left} = Xw + b_{left}^{(0)} = [0.1, 0.5] \begin{bmatrix} 0.15 & 0.20 \\ 0.25 & 0.30 \end{bmatrix} + [0.34, 0.45] = [0.48, 0.62]$, and $\ell_{left} = \frac{1}{2}[(0.48 - 0.01)^2 + (0.62 - 0.99)^2] = 0.1789$.
  - $y_{right} = Xw + b_{right}^{(0)} = [0.1, 0.5] \begin{bmatrix} 0.15 & 0.20 \\ 0.25 & 0.30 \end{bmatrix} + [0.36, 0.45] = [0.50, 0.62]$, and $\ell_{right} = \frac{1}{2}[(0.50 - 0.01)^2 + (0.62 - 0.99)^2] = 0.1885$.
  - Then we can compute the derivative of loss with respect to the bias $b_{11}^{(0)}$ as $\frac{\partial \ell}{b_{11}^{(0)}} = \frac{\ell_{right} - \ell_{left}}{2\epsilon} = \frac{0.1885 - 0.1789}{0.02} = 0.48$.

- For $b_{12}^{(0)}$, after computation, we have $\frac{\ell}{b_{12}^{(0)}} = \frac{0.18485-0.19225}{0.02} = -0.37$. Hence we have $\frac{\partial \ell}{\partial b} = [0.48, -0.37]$ and it is consistent with the results we have before.

After verified the weights and biases, we now proceed to verify the input matrix as below:

- There are two inputs in our example, the $x_1^{(0)} = 0.1$ and $x_2^{(0)} = 0.5$. For $x_1^{(0)}$, we define $x_{left} = x_1^{(0)} - \epsilon = 0.09$, $x_{right} = x_1^{(0)} + \epsilon = 0.11$ and X_{left}, X_{right} are the corresponding matrices. Then we have:
  - $y_{left} = X_{left}w + b^{(0)} = [0.09, 0.5] \begin{bmatrix} 0.15 & 0.20 \\ 0.25 & 0.30 \end{bmatrix} + [0.35, 0.45] = [0.4885, 0.618]$, and $\ell_{left} = \frac{1}{2}[(0.48 - 0.01)^2 + (0.62 - 0.99)^2] = 0.183673125$.
  - $y_{right} = X_{right}w + b^{(0)} = [0.11, 0.5] \begin{bmatrix} 0.15 & 0.20 \\ 0.25 & 0.30 \end{bmatrix} + [0.35, 0.45] = [0.4915, 0.622]$, and $\ell_{right} = \frac{1}{2}[(0.4915 - 0.01)^2 + (0.622 - 0.99)^2] = 0.183633125$.
  - Then we can compute the derivative of loss with respect to the input $x_1^{(0)}$ as $\frac{\partial l}{\partial x_1^{(0)}} = \frac{0.183633125-0.183673125}{0.02} = -0.002$.
- Similarly, for $x_2^{(0)}$, after computation, we have $\frac{\partial \ell}{\partial x_2^{(0)}} = \frac{0.183747625-0.183567625}{0.02} = 0.009$. Hence we will have $\frac{\partial \ell}{\partial X} = [-0.002, 0.009]$, and the result is consistent with our results above.

By performing the numerical verification as cross-validation, we can confirm that our inductions and computations are correct. With knowing the forward pass and backward pass, we now proceed to investigate how convolution and other operations work.

3.1.2 Convolution

Definition In fully connected layers, the input is always a one-dimensional vector. However, images are usually stored as a multi-dimensional matrix and have implicit spatial structures. For example, the eyes are always on top of the nose, etc. These properties are not well expressed using a fully connected layer. Hence we use the convolutional layers to preserve these properties. For example, assume we have a single channel input matrix $A_{3\times3}$ and single filter matrix $B_{2\times2}$, and

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}, B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

Then, we slide the filter B with a unit stride, i.e. move one column or one row at a time. If we use a_{ij} and b_{ij} to denote the element in A and B at the (i, j) location. Then we can obtain the output of the convolutional layer with the following steps:

- At the beginning, the 2×2 filter is placed at the upper left corner. At this time, we perform the dot product and we will have $y_{11} = a_{11}b_{11} + a_{12}b_{12} + a_{21}b_{21} + a_{22}b_{22}$.

- Then we slide the filter across the width for a unit stride, i.e. we move the slide to the upper right corner. At this time, we perform the dot product and we will have $y_{12} = a_{12}b_{11} + a_{13}b_{12} + a_{22}b_{21} + a_{23}b_{22}$.
- Then we found that there is no more values on the right side, so we start to slide the filter on the next row. At this time, we start at the bottom left corner and we can obtain that $y_{21} = a_{21}b_{11} + a_{22}b_{12} + a_{31}b_{21} + a_{32}b_{22}$.
- Then we again slide the filter to the right side, i.e. the bottom right corner and we obtain that $y_{22} = a_{22}b_{11} + a_{23}b_{12} + a_{32}b_{21} + a_{33}b_{22}$.

After these steps, we found that we get four outputs, and we can obtain the final output if we place the output values to corresponding locations, i.e. the value we computed at the upper left corner is placed at the upper left corner and so on so forth. Hence, in this example, we get a $(1, 2, 2)$ output matrix $C = \begin{bmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \end{bmatrix}$

More generally, we can formalize the input, the filters and the output of a convolutional layer as below:

- The input is a (N, C, H, W) tensor, where N is the number of the input matrix in a single batch, C is the channel of the matrix, H and W are the height and width of the input matrix. For example, 10 coloured images with the size $(224, 224)$ can be represented as a $(10, 3, 224, 224)$ tensor.
- The filters is a (K, C, H_f, W_f) tensor, where K is the number of filters, C is the channel of the filters and it will always be identical to the channel of the input matrix. H_f and W_f are the height and width of the filters.
- The output is a (N, K, H_{out}, W_{out}) tensor.
- The stride that we use to slide the filters are denoted as S.

With these notations, we can compute the output of a convolution layer with the algorithm below:

Algorithm 1: Direct Convolution Computation

Input: A (N, C, H, W) tensor as data and a (K, C, H_f, W_f) tensor as filter
Result: The output of a convolution layer, as a tensor of the shape (N, K, H_{out}, W_{out})
Let output be a zero matrix of the shape (N, K, H_{out}, W_{out})

```
while n < N do
    while o_y < H_out do
        while o_x < W_out do
            while o_k < K do
                while o_h < H_f do
                    while o_w < W_f do
                        while o_c < C do
                            prod ←
                              input[n][o_c][o_y * S + o_h][o_x * S + o_w] · filter[o_k][o_c][o_h][o_w]
                              output[o_k][o_c][o_y][o_x] += prod
                        end
                    end
                end
            end
        end
    end
end
```

Though the convolution operation can be computed by the above algorithm, we can still use matrix multiplication to perform such computation as suggested by [DV16]. The benefits of using matrix multiplication are two-fold:

- We have already gotten two laws for computing the differentiation of linear transformation. If we can define the convolution operation as $g(x) = AX + B$ (i.e. matrix multiplication), we could easily reuse the two laws and get the derivative of the loss value with respect to the filter and input.
- We are about to study how to compute the forward pass of Deconv operation and that operation can be easily defined with the matrix multiplication form of the convolution operation. We will see this in *Section 3.2.1 Deconv*.

Hence, in the below computation of the forward pass and backward pass of the convolution operation, we will show how to convert it into a matrix multiplication.

Forward Pass Recall that the input X, the filter W and the expected output Y we have in the above example.

$$X = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}, W = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}, Y = \begin{bmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \end{bmatrix}$$

If we unroll the input and the output into vectors from left to right, top to bottom, we can also represent the filters as a sparse matrix where the non-zero elements are the ele-

ments in the filters. For example, We can unroll the input and output in our case as $X^*_{9\times 1} = [a_{11}, a_{12}, a_{13}, a_{21}, a_{22}, a_{23}, a_{31}, a_{32}, a_{33}]^T$ and $Y^*_{4\times 1} = [y_{11}, y_{12}, y_{21}, y_{22}]^T$. Then we will want a 4×9 matrix W^* such that $Y^* = W^* X^*$. From the direct computation of convolutional layers, we can transform the original filters W into

$$W^* = \begin{bmatrix} b_{11} & b_{12} & 0 & b_{21} & b_{22} & 0 & 0 & 0 & 0 \\ 0 & b_{11} & b_{12} & 0 & b_{21} & b_{22} & 0 & 0 & 0 \\ 0 & 0 & 0 & b_{11} & b_{12} & 0 & b_{21} & b_{22} & 0 \\ 0 & 0 & 0 & 0 & b_{11} & b_{12} & 0 & b_{21} & b_{22} \end{bmatrix}$$

Then we can verify that

$$W^* X^* = \begin{bmatrix} b_{11} & b_{12} & 0 & b_{21} & b_{22} & 0 & 0 & 0 & 0 \\ 0 & b_{11} & b_{12} & 0 & b_{21} & b_{22} & 0 & 0 & 0 \\ 0 & 0 & 0 & b_{11} & b_{12} & 0 & b_{21} & b_{22} & 0 \\ 0 & 0 & 0 & 0 & b_{11} & b_{12} & 0 & b_{21} & b_{22} \end{bmatrix} \times [a_{11}, a_{12}, a_{13} \cdots, a_{33}]^T = Y^*$$

Backward Pass From the forward pass, we converted the filters into a sparse matrix W^*, and we found that the convolution is also a linear operation, i.e. $Y^* = W^* X^*$. Similar to the backward pass of fully connected layers, we can directly compute the gradient of the loss value with respect to the weight matrix as $\frac{\partial \ell}{\partial W^*} = \nabla^{(i)} (X^*)^T$ (Using the Law 2). Hence, we can update the weight matrix in convolutional layers as $(W^*)^{new} = (W^*)^{old} - \epsilon \nabla^{(i)} (X^*)^T$ and it is the same as in fully connected layers.

Besides, we will need to compute the gradient that we want to pass to previous layers, i.e. the gradient of the loss value with respect to the input matrix. We will have $\frac{\partial \ell}{\partial X^*} = \frac{\partial \ell}{\partial Y^*} \frac{\partial Y^*}{\partial X^*} = (W^*)^T \nabla^{(i)}$ (Using the Law 1).

Example Here we will show how the unrolling process works for convolution operation and how to perform the forward pass in the direct and matrix multiplication methods. Since the backward pass is identical to fully connected layers, we will not compute the backward pass in this example.

We assume that we have an input X, the filter W as

$$X = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, W = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

Direct Computation To compute the output directly, we will have to slide the filter W from left to right, from top to bottom. We will have $1*1+2*2+4*3+5*4 = 37$, $2*1+3*2+5*3+6*4 = 47$, $4*1+5*2+7*3+8*4 = 67$, $5*1+6*2+8*3+9*4 = 77$ in the upper left, upper right, bottom left and bottom right corners. Then, we can get the output as

$$Y = \begin{bmatrix} 37 & 47 \\ 67 & 77 \end{bmatrix}$$

Matrix Multiplcation With matrix multiplication approach, we need to unroll the filter and

input matrices into

$$W^* = \begin{bmatrix} 1 & 2 & 0 & 3 & 4 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 0 & 3 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 2 & 0 & 3 & 4 & 0 \\ 0 & 0 & 0 & 0 & 1 & 2 & 0 & 3 & 4 \end{bmatrix}, X^* = \begin{bmatrix} 1 & 2 & \cdots & 9 \end{bmatrix}^T$$

Then we can compute the output directly by $Y^* = W^*X^* = [37, 47, 67, 77]^T$. By reshaping Y^*, we can easily obtain the desired output matrix $Y_{2\times 2}$.

Since the backward process will be identical to what we did in *Section 3.1.1 Fully Connected* if we perform the forward pass in a matrix multiplication way, we will omit the examples here.

3.1.3 Pooling

Definition Pooling layer is another important component of the convolutional neural network, which is used to reduce the spatial size of the intermediate results to reduce the amount of parameters. The most common kind of pooling is max pooling, which splits the input in non-overlapping patches, and output the maximum value of each patch. For example, assume the input that we have is,

$$X = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Assume that we have a 2×2 max pooling layer, then the pooling layer will first split the input matrix into small 2×2 patches: the upper left, the upper right, the bottom left and the bottom right corners. Then in each patch, we can find the maxmium values are $[5, 6, 8, 9]$, hence, the output of the max pooling layer is $P = \begin{bmatrix} 5 & 6 \\ 8 & 9 \end{bmatrix}$.

Forward Pass In the forward pass, only the maximum value in each patch will be left, and all other values will be dropped. Similar to the computation process of convolutional layers, we also have a spatially small sliding window called the kernel, in which the largest value will be kept. Assume that the input of the pooling layer is a (H, W) matrix and if we set the height and width of the kernel as (K_h, K_w) and we use a stride S to slide the kernel from left to right, from top to bottom. Then in each kernel, we select the maximum value as output. We can induce the output shape of a pooling layer as $(\lfloor \frac{H-K_h}{S} \rfloor + 1, \lfloor \frac{W-K_w}{S} \rfloor + 1)$.

The forward computation can be achieved with the following algorithm:

Algorithm 2: Algorithm for Computing Pooling Output

Input: A (H, W) tensor as data and a (K_h, K_w) tensor as filter
Result: The output of a pooling layer
Let output be a matrix of the shape $(\lfloor \frac{H-K_h}{S} \rfloor + 1, \lfloor \frac{W-K_w}{S} \rfloor + 1)$ filled with $-\infty$
while $h < \lfloor \frac{H-K_h}{S} \rfloor + 1$ **do**
  while $w < \lfloor \frac{W-K_w}{S} \rfloor + 1$ **do**
    while $i_h < K_h$ **do**
      while $i_w < K_w$ **do**
        $h_{index} \leftarrow h * S + i_h$
        $w_{index} \leftarrow w * S + i_w$
        if *output[h][w] < input[*h_{index}*][*w_{index}*]* **then**
          output[h][w] = input[h_{index}][w_{index}]
        end
      end
    end
  end
end

Backward Pass In the backward pass, since there is no parameter that need to be adjusted during the training, we only need to compute the gradient that we want to pass to the previous layer. The backward pass of pooling layer can be computed by two parts:

- If the value x_{kj} in the input matrix is kept, then we have $\frac{\partial \ell}{\partial x_{kj}} = \frac{\partial \ell}{\partial y}\frac{\partial y}{\partial x_{kj}} = \frac{\partial \ell}{\partial y} = \nabla_{kj}^{(i)}$.
- If the value x_{kj} in the input matrix is dropped, then we have $\frac{\partial \ell}{\partial x_{kj}} = \frac{\partial \ell}{\partial y}\frac{\partial y}{\partial x_{kj}} = 0$.

3.1.4 Relu Layer

Definition From the above sections, we have found that both the fully connected layer and convolutional layer can be categorized into linear functions. However, the reality is usually non-linear and cannot be fitted by compositing linear functions. Hence, we need to bring some non-linearity into the neural network architecture. Among the many non-linear functions, rectifier linear unit (ReLu) is the most commonly used non-linearity. It is defined as a function that $f(x) = max(0, x)$

Forward Pass The input of a ReLu layer is a tensor with any dimensions, and we perform element-wise ReLu operation on the input, i.e. it will be kept if it is positive, otherwise it will be set to zero.

Backward Pass Similar to pooling layers, we can separate the backward pass of ReLu layer into two parts:

- In the location (i, j), if the value is kept unchanged, then the derivatives will be $\frac{\partial \ell}{\partial x_{ij}} = \frac{\partial \ell}{\partial y}\frac{\partial y}{\partial x_{ij}} = \nabla_{ij}^{(i)}$.
- Otherwise, if $x_{i,j} \leq 0$, then $\frac{\partial \ell}{\partial x_{ij}} = \frac{\partial \ell}{\partial y}\frac{\partial y}{\partial x_{ij}} = \nabla^{(i)} \times 0 = 0$.

3.2 Interpretation Model

We use a deconvolutional neural network to interpret and understand the result of our classification model. Here they are not used for any training purpose, but only as a probe of the trained classification model. Therefore, in this section, we only need to have the forward pass of a deconvolution neural network and do not need to compute the backward pass.

3.2.1 Deconv

Definition In *Section 3.1.2 Convolution*, we unrolled the filter from a 2×2 matrix into a 4×9 matrix, so that we can perform the convolution by matrix multiplication. After the convolution operation, the input data changes from a 3×3 matrix to a 2×2 matrix. The deconv operation is defined as the inverse of the convolution operation, i.e. change the input data from a 2×2 matrix into an output matrix with the shape 3×3 in our example. The deconv operation does not guarantee that we will have the same values in the output as the original matrix. Below we will show how it is computed in the forward pass.

Forward Pass When computing the forward pass of deconv operation, we can simply transpose the unrolled filters matrix, for example, it will be a 4×9 matrix in our case. After the transpose, we can define the deconv operation as $X = (W^*)^T Y$, i.e. we use the transposed, and unrolled filter matrix to multiply the output of the convolution operation.

Example We assume that we have an input Y(exactly the same with the output of the convolution operation in our previous example, hence we will use Y as the notation for this input) and the same filter W as

$$Y = \begin{bmatrix} 37 & 47 \\ 67 & 77 \end{bmatrix}, W = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

Then we want to get a 3×3 matrix as the output of the deconv operation. Recall that we unrolled the filter into the matrix as

$$W^* = \begin{bmatrix} 1 & 2 & 0 & 3 & 4 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 0 & 3 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 2 & 0 & 3 & 4 & 0 \\ 0 & 0 & 0 & 0 & 1 & 2 & 0 & 3 & 4 \end{bmatrix}$$

We can compute the desired matrix by performing transpose on the filter matrix first, and then multiply it with our input. We will have

$$X = (W^*)^T Y_{4\times 1} = [37, 121, 94, 178, 500, 342, 201, 499, 308]^T$$

Then we can reshape it back into a 3×3 matrix as $X_{3\times 3} = \begin{bmatrix} 37 & 121 & 94 \\ 178 & 500 & 342 \\ 201 & 499 & 308 \end{bmatrix}$

As we see in this example, the deconv operation does not guarantee that we will have the same input of convolution operation, but just guarantee we will have a matrix with the same shape as the input of convolution operation. Since the entries may exceed the maximum light

intensity, i.e. 255, when we are visualizing the deconv result, we will need to renormalize every entry into the range of $[0, 255]$.

3.2.2 Max-Unpooling

Unfortunately, as some inputs are being dropped in the pooling layer (because only the maximum value will be kept), we cannot fully inverse the max-pooling operation. However, if we have the position of each maximum value when performing max pooling, then we can simply put the maximum value back to its original position. After putting them back, we can set values at other positions to be 0.

Example As in the pooling section, we have an input matrix $X = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ and the corresponding output $P = \begin{bmatrix} 5 & 6 \\ 8 & 9 \end{bmatrix}$. Besides these, we will know the indices of the maximum value in each region, which are $(1, 1)$, $(1, 2)$, $(2, 1)$ and $(2, 2)$.

In the unpooling process, we first put the maximum values back to its position, and fill other positions with 0. We will get the output as $\begin{bmatrix} 0 & 0 & 0 \\ 0 & 5 & 6 \\ 0 & 8 & 9 \end{bmatrix}$

3.2.3 UnReLu

In [MF14], they passed the reconstructed data into the ReLu non-linearity during the interpretation process. They claimed that they also tried using the binary mask imposed by the forward ReLu operation, but the resulting visualizations were significantly less clear. Hence, in this report, we will also use the ReLu function as the inverse of ReLu, as suggested by their article.

4 Running Example

In *Section 3 Approach*, we already have several examples describing how convolution and deconvolution network works. Now that it is clear that how to compute the backward and forward pass of those networks, we will provide a running example of the whole process, including training, evaluating and interpreting convolutional neural networks.

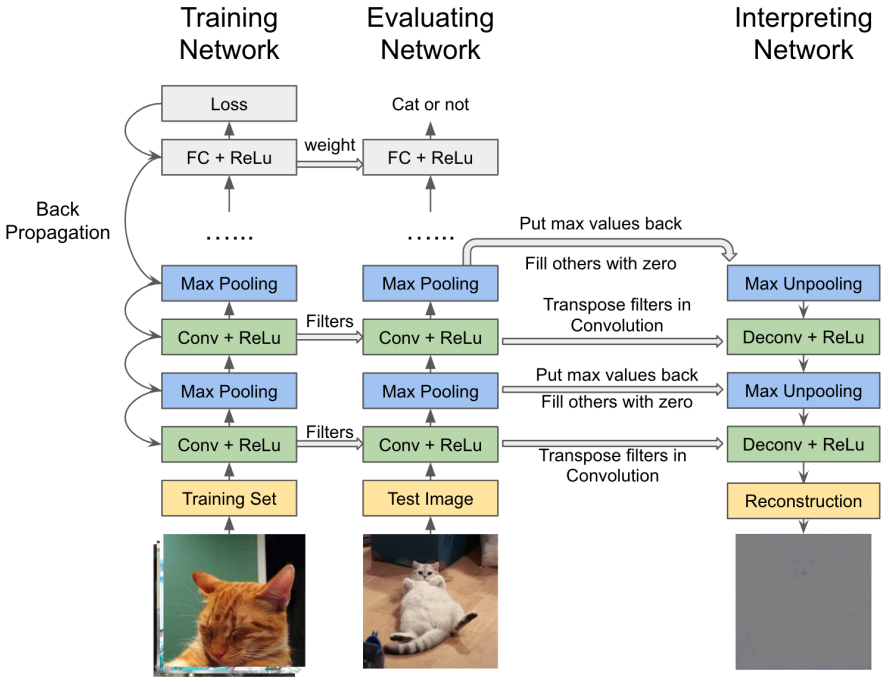


Figure 4.1: Illustration of an example workflow in this work. We attached an inverse neural network, called the deconv net to the convolutional neural network. The deconv net will reconstruct an approximated image from the activations in the convolutional neural networks.

In Fig 4.1, we illustrate the training process on the left, where we first prepare several images as the training dataset. After that, we perform the forward pass of our neural network on the given training dataset, and compute the loss value. After that, we start to do backpropagation from the loss value to the very first convolution layer. These process will be performed several times until we are satisfied with the loss value.

In the middle, we show how we evaluate our neural networks and the training process. In the evaluation, we first fill the parameters in our neural network with the parameters that we have gotten during the training. Then, with a given test image, we only need to perform the forward pass once, and our neural network will tell us if it is a cat. We can then compare the result with ground truth.

On the right is the visualization and interpretation process of our neural network, which consists of two components: the max-unpooling operation and the deconvolution operation. After evaluating the model, we first convert the parameters to the deconvolution network by doing two steps:

- For the max-pooling layers, we put the max values back to their position and fill other positions with 0.
- For the convolutional layers, we transpose the filter matrix and assign the transposed matrix to deconvolution layer.

After that, we can start to visualize the neural network by performing a forward pass on top of the activations in the convolutional layer and we will get the reconstructed image after the forward pass.

In the above example, we showed how those components, including convolution, fully connected, pooling and deconvolution works in this report. In the next several sections, we will show how we implement those components, and how we perform the experiments on the given dataset.

5 Implementation

In our implementation, we focus on three main tasks: construct the neural networks, perform the training, evaluating and visualizing processes and export the trained weight to the persistent storage (i.e. the hard disk). We have made the following modules to achieve these goals:

- Core. In this module, we implement a base class for all the parameters that need to be updated during training. A parameter includes two components: the tensor that saves the actual data, and the gradient that saves the derivatives of the loss with respect to the parameter for updating.
- Layers. We implement all the needed layers in this module, including the fully connected layer, the convolutional layer, ReLu and Dropout layer, etc. All these layers are Python classes that are extended from a base class, which requires the subclasses to implement a *forward* and a *backward* function.
- Losses. We implement the needed cross-entropy loss in this module. The loss function is implemented as a Python function that has two inputs, *predicted* and *ground truth*. Then the function needs to return two values, the *loss value*, which measures the distance between the ground truth and the predicted output, and the *gradient*, which calculates the derivatives of loss value with respect to the predicted output.
- Net. Net is a class that stacks several different layers, and provides three functions: *forward*, *backward* and *update*. The forward function will compute the output of the forward pass from the beginning of the stacked layers, while the backward function will first reverse those layers and then compute the backward pass from the end of the given layers. The update function simply updates all those parameters in a neural network at once.
- Optimizer. The SGD optimizer is implemented in this module. The optimizer receives a parameter from the *Core* module, computes the next value by $new = old - \epsilon\nabla$ where ϵ is the preset learning rate, and ∇ is the computed derivative of the loss with respect to the parameter.
- Learner. We perform the actual training process inside the *Learner* module. A learner receives a user-defined neural network architecture, a training dataset, an optimizer and some other hyperparameters such as batch size. Then the learner will read the training dataset batch by batch, and in each batch, the learner will call the forward function of the given neural network architecture on the batch, compute the loss value and then perform

the backward pass. After the backward pass in each batch, the learner will update all the parameters in the network.

In order to understand how our implementation works, we will show some important code snippets in this section. We will start by showing how fully connected layer (it is called linear layer in our implementation) works in the Listing 5.1.

```
class Linear(Layer):
    def __init__(self, name, input_dim, output_dim):
        super().__init__(name)
        weight = np.random.randn(
            input_dim, output_dim) * np.sqrt(1/input_dim)
        bias = np.random.randn(output_dim) * np.sqrt(1/input_dim)
        self.type = 'Linear'
        self.weight = self.build_param(weight)
        self.bias = self.build_param(bias)

    def forward(self, input):
        self.input = input
        return np.matmul(input, self.weight.tensor) + self.bias.tensor

    def backward(self, in_gradient):
        self.weight.gradient = np.matmul(self.input.T, in_gradient)
        self.bias.gradient = np.sum(in_gradient, axis=0)
        return np.matmul(in_gradient, self.weight.tensor)
```

Listing 5.1: Implementation of Linear Layer in Our Implementation

The class of fully connected layer extends from the base class *Layer*, and implement a *init* function that initializes the weight and bias in this layer, a *forward* function that computes the output with the given input, and a *backward* function that computes the derivatives of the loss value with respect to the weight, bias and input of this layer. In the backward function, the derivatives of the weight and bias will be saved to the parameters themselves, while the derivatives of the input will be returned and will be received by previous layers to perform the chain rule. Other layers, such as the convolutional, max-pooling and ReLu layers are implemented in the same way.

Then we implement a *Net* class that stacks several layers. This class will also provide a forward function that calls the forward method from the very first layer to the last layer one by one. A backward function is also provided and it will call the backward method from the last layer to the first layer. The implementation is shown in Listing 5.2.

```
class Net:
    def __init__(self, layers):
        super().__init__(layers)

    def forward(self, input, return_indices=False):
        pool_indices = [None] * len(self.layers)
        output = input
        for index, layer in enumerate(self.layers):
            if isinstance(layer, MaxPool2D) and layer.return_index:
                output, max_indices = layer.forward(output)
```

```
                pool_indices[index] = max_indices
            else:
                output = layer.forward(output)
            output_intermediate_result(layer.name, output, 'data', layer)
        if return_indices:
            return output, pool_indices
        return output

    def backward(self, in_gradient):
        out_gradient = in_gradient
        for layer in self.layers[::-1]:
            out_gradient = layer.backward(out_gradient)
            output_intermediate_result(layer.name, out_gradient, 'gradient
    ', layer)
        return out_gradient
```

Listing 5.2: Implementation of the *Net* class

With these layers and the *Net* class, we can stack the layers by providing a list of instances of different layers, as shown in Listing 5.3.

```
model = Net([
    Conv2D('conv_1', (1, 28, 28),
           n_filter=32,
           h_filter=3,
           w_filter=3,
           stride=1,
           padding=0),
    ReLu('relu_1'),
    Conv2D('conv_2', (32, 26, 26), 64, 3, 3, 1, 0),
    ReLu('relu_2'),
    MaxPool2D('maxpool_1', (64, 24, 24), size=(2,2), stride=2),
    Dropout('drop_1', 0.25),
    Flatten('flat_1'),
    Linear('fc_1', 9216, 128),
    ReLu('relu_3'),
    Dropout('drop_2', 0.5),
    Linear('fc_2', 128, 10),
])
```

Listing 5.3: Stacking Layers with the *Net* class

After defining the neural network architecture, we can create our learner and let the neural network fit the training dataset, as shown in Listing 5.4. After the training process, the parameters will be frozen inside the model (i.e they will not be updated anymore), which should be an instance of the *Net* class, and it can be easily exported with the *export* function provided by the *Net* class.

```
learner = Learner(model, cross_entropy_with_softmax_loss,
                  SGDOptimizer(lr=0.1, momentum=0.9))

learner.fit(x_train, y_train, epochs=5, batch_size=1024)

```

```
model.export('saved.tnn')
```

Listing 5.4: Creation of Learner and the Use of *fit* Function

5.1 Correctness Validation

After implementing all these needed components, we need to validate that these are correct. In order to confirm the correctness, we added test cases to every layer that compare the results between our implementation and the PyTorch version. Besides these, we also designed a large test case that performs an actual training process with random values, and we compare the weight and bias between our implementation and the PyTorch version in every epoch.

More specifically, in every layer, we first compare the output between the two implementations with the same input and weight value. The test case for the forward computation in the fully connected layer is shown in Listing 5.5 as an example.

```
class TestLinearLayer(unittest.TestCase):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        batch_size = 3
        in_features = 5
        out_features = 2
        self.data = np.random.randn(batch_size, in_features)
        self.forward_weight = np.random.randn(out_features, in_features)
        self.forward_bias = np.random.randn(out_features)
        self.torch_linear = torch_linear(in_features, out_features)
        self.tnn_linear = tnn_linear('test', in_features, out_features)
        self.gradient = np.random.randn(batch_size, out_features)
        self.torch_input = torch.from_numpy(self.data)
        self.torch_input = self.torch_input.view(self.torch_input.size(0),
    -1)
        self.torch_input.requires_grad = True
        self.torch_input.retain_grad()

    def test_forward(self):

        self.torch_linear.weight = torch.nn.Parameter(
            torch.from_numpy(self.forward_weight))
        self.torch_linear.bias = torch.nn.Parameter(
            torch.from_numpy(self.forward_bias))

        self.tnn_linear.weight.tensor = self.forward_weight
        self.tnn_linear.bias.tensor = self.forward_bias

        self.torch_output = self.torch_linear(self.torch_input)
        self.tnn_output = self.tnn_linear(self.data)

        self.assertTrue(
            np.absolute(self.torch_output.detach().numpy() -
```

```
                            self.tnn_output < EPSILON).all())
```

Listing 5.5: Test Case for Forward Computation of Fully Connected Layer

After validating the forward pass, we need to validate the backward pass as well. In convolutional and fully connected layer, we need to compare three values: the derivatives of the loss value with respect to the weight, bias and input. In other layers that do not have weight and bias in them, we only need to compare the loss value with respect to the input. The test case for backward computation in the fully connected layer is shown in Listing 5.6 as an example.

```
class TestLinearLayer(unittest.TestCase):
    def __init__(self, *args, **kwargs):
        # the same with the forward test case
    def test_backward(self):
        self.test_forward()
        output_grad = self.tnn_linear.backward(self.gradient)
        self.torch_output.backward(torch.from_numpy(self.gradient))
        self.assertTrue(
            np.absolute((self.torch_input.grad - output_grad) < EPSILON).
    all())
        self.assertTrue(
            np.absolute((self.torch_linear.weight.grad.numpy() -
                         self.tnn_linear.weight.gradient) < EPSILON).all()
    )
        self.assertTrue(
            np.absolute((self.torch_linear.bias.grad.numpy() -
                         self.tnn_linear.bias.gradient) < EPSILON).all())
```

Listing 5.6: Test Case for Forward Computation of Fully Connected Layer

In addition, we also performed a large test. In that test, we build a small neural network, and give it with some random input, and then perform the training process. During the training process, we compare the two models (one is PyTorch implementation, one is ours) in every epoch.

6 Experiment

In this work, we reproduced the results from [MF14]. Since our implementation is significantly slower than other mature frameworks, we decided to use smaller dataset, smaller neural network architecture and a binary classifier to perform the reproduction.

6.1 Classification Settings

Dataset We randomly pick 600 cat images from *dogs-vs-cats* dataset [PVZJ12], which is a large dataset consisted of 25,000 images of cats and dogs. Then we randomly pick 600 non-cat images from the Imagenet dataset [DDS+09]. All these images are resized to $(224, 224)$ with 3 colour planes. Then we randomly split the dataset into two pieces: 90% (1080) of the data are used for training and others are used for validation.
Network Architecture We designed a smaller neural network architecture based on VGG-16 net. In our network, there are four convolutional blocks, which is consisted of two to three convolutional layers and a pooling layer as feature extractor and three fully connected layers as the classifier. The architecture of our classification model is illustrated in 6.1.

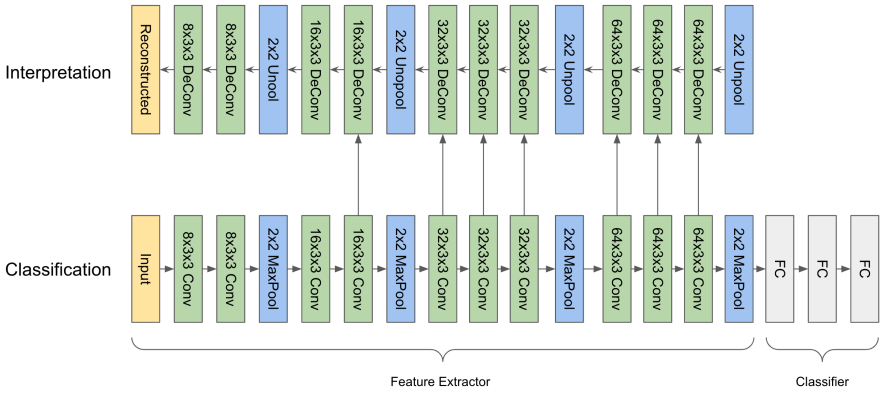


Figure 6.1: Illustration of the Network Architecture in Our Experiments. Note that some components, such as the Dropout and the ReLu after each convolution and fully connected layer, are not shown in the figure. Here $n \times h \times w$ Conv(Deconv respectively.) refers to many (n) $h \times w$ filters in a Conv (Deconv respectively.) layer. $h \times w$ Maxpool (Unpool respectively.) refers to the size of the filters in pooling (unpooling) layer.

Training Settings During training, we first renormalize the images from $[0, 255]$ integers to $[0, 1]$ floating numbers, and then feed into the classification model with a mini-batch size of 36. After each batch, we apply SGD to update the parameters, starting with a learning rate of 10^{-3}, in conjunction with a momentum term of 0.9. Dropout is used in the fully connected layers with a rate of 0.5. All weights and biases are randomly initialized from a normal distribution.
Results The classifier that we trained achieved 85% accuracy on the validation set, i.e. 102 images are correctly classified in 120 images. On the test dataset, which consists of 20 cat images, 65% (13) are correctly classified as a cat.

6.2 Feature Visualization

As seen in Fig 6.1, we attached a deconv net to the output of some convolutional layers, so that the deconv net will reconstruct the approximate version of the activations in the convolutional neural network. Some results are shown in Fig 6.2

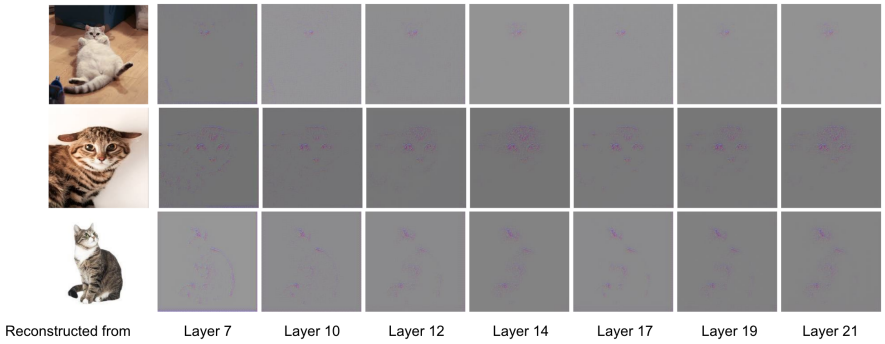


Figure 6.2: Illustration of the Reconstructed Images.

In the illustration, we show the image reconstructed from the output of certain convolutional layers. From what we saw here, we found that the first several layers can extract some general features, such as the boundaries. Then in the next several layers, we found that the features are becoming more and more focused on parts of cat object, for example, in the 21th layer of the first image, the features are more concentrated on the head of the object. This also applies to the second image, where the first several layers can extract the head of the cat, and the last several layers can extract a smaller region - the eyes of the cat. These results show that our classifier mainly depends on the head region of the object to determine if it is a cat.

6.3 Comparison between Ours and PyTorch Implementation

We compared our results with a PyTorch implementation [Huy], which uses the full VGG-16 network and are trained on the Imagenet [DDS+09] dataset. Their results are illustrated in

Fig. 6.3

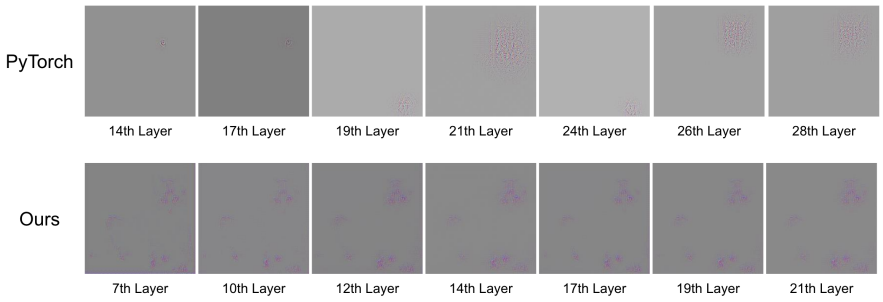


Figure 6.3: Illustration of reconstructed image from intermediate activations from PyTorch and Ours Implementation. The nth Layer refers to the layer where we start to reconstruct.

From their visualizations, we found that the classifier also primarily focused on the head region of the cat image. However, their classifier could not only activate the head region in deeper convolution layers (in 24, 26, 28 Layers), but also some smaller regions, such as the left eye (in 14 Layer) and the right eye (in 17 Layer). Some other regions are also activated such as feet (in 19 and 21 Layers). We think there might be several reasons leading to the differences:

- The PyTorch version uses a full VGG-16 network, and are equipped with 138 million parameters while ours uses a reduced version, with 68 million parameters. It is easier for our reduced neural network to be trained on the dataset, as there are fewer parameters to update. However, these parameters may not be enough to activate the smaller regions.
- We only use 1080 images as the training set while the PyTorch version uses 14 million images.

In summary, the advantages of PyTorch version is that they have more parameters and larger dataset, and hence should work not only on cat images, but also other categories. However, even though our network is significantly smaller, we also revealed that the classifier mainly focuses on head regions to perform the classification.

Bibliography

[DDS+09] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[DV16] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*, 2016.

[Huy] Huybery. huybery/visualizingcnn.

[MF14] D Matthew and R Fergus. Visualizing and understanding convolutional neural networks. In *Proceedings of the 13th European Conference Computer Vision and Pattern Recognition, Zurich, Switzerland*, pages 6–12, 2014.

[PVZJ12] Omkar M Parkhi, Andrea Vedaldi, Andrew Zisserman, and CV Jawahar. Cats and dogs. In *2012 IEEE conference on computer vision and pattern recognition*, pages 3498–3505. IEEE, 2012.

[SZ14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.