tinyml

Xiaozhe Yao

Jul 16, 2021
tinyml is a small framework for writing neural networks for educational purposes. It is written in pure Python with some help from third party libraries, such as numpy, tqdm, etc. I hope this is a good source for learning neural networks and deep learning.

There might be some errors and deficiencies, and these are all my fault. If you meet any of them, please kindly leave your comments at GitHub Issue Tracker. If you are not comfortable with publicly accessible issues, please drop an email to askxyao@gmail.com.
I wrote tinyml for learning purpose, and I found it is useful and interesting to figure out what is a deep learning framework doing under their apis. I hope this project can be useful for other students to understand how deep learning works.

In tinyml, 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 − ε∇` 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.
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 \textit{weight} and \textit{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 \( \alpha \) 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 \( \epsilon \) 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 \( i \)th layer, we can receive the gradient of the loss \( \ell \) with respects to the output of \( i \)th layer, i.e. \( \frac{\partial \ell}{\partial y^{(i)}} \) is known to us.
- Since the output of \((i-1)\)th layer is the input of the \( i \)th layer, we have \( \frac{\partial \ell}{\partial x^{(i)}} = \frac{\partial \ell}{\partial y^{(i-1)}} \).
Having these two bases, we could compute the gradient of the loss $\ell$ with respect to the weight and input of every layer by applying chain rules. For example, $\frac{\partial \ell}{\partial w(i)} = \frac{\partial \ell}{\partial y(i)} \frac{\partial y(i)}{\partial w(i)}$ 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.

**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 $\epsilon$ 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 $\epsilon$, 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.
In the below sections, we will by default use the following notations:

- \( x^{(i)}_j \) is the \( j \)th input of the \( i \)th layer. In fully connected layers, all the inputs form a vector, thus we can use \( x^{(i)}_j \) to denote them. However, in convolutional layers, the inputs form a 2d matrix, and in that case, we will use \( x^{(i)}_{kj} \) to denote the individual input. The formed vectors (in fully connected layers) or matrices (in convolutional layers) will be represented by \( X^{(i)} \).

- \( y^{(i)}_j \) is the \( j \)th output of the \( i \)th layer. If the \( i \)th layer is the last layer, there might be some corresponding ground truths. In this case, the ground truths will be denoted as \( y^{(i)}_j \). We will use \( \hat{Y}^i \) to denote the vector or matrix that contains all the outputs of the \( i \)th layer. Correspondingly, the vector or matrix of the ground truth will be denoted by \( Y^{(i)} \).

- \( \ell \) is the value of loss function. In our classification model, we use the cross-entropy loss and it is defined as \( \ell = -\sum_i^n y^{(i)}_j \log(\hat{y}^{(i)}_j) \) 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 \( i \)th layer. Formally, we have \( \nabla^{(i)} = \frac{\partial \ell}{\partial \hat{Y}^{(i)}} \). Since the output of the \( i \)th 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 \( \nabla^{(i)}_{kj} \) 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 \( \epsilon \) is a small enough real number.
4.1 Linear Transformation of Matrices

**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 $f(Y) : \mathbb{R}^{m \times n} \rightarrow \mathbb{R}$ and a linear transformation $g(X) : \mathbb{R}^{p \times n} \rightarrow \mathbb{R}^{m \times n}, Y = g(X) = AX + B,$ 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 composed 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 $y_{kl} = \sum_{ij} a_{kj}x_{il} + b_{kl}$. Hence we have $\frac{\partial y_{kl}}{\partial x_{ij}} = \sum_{ij} \frac{\partial a_{ij}x_{il}}{\partial x_{ij}} = \delta_{ij}$ (2). Here $\delta_{ij}$ is defined as $\delta_{ij} = 1$ when $l = j$, otherwise $\delta_{ij} = 0$. Intuitively, we know that for the single pair $(x_{ij}, y_{kl})$, there is a relation $y_{kl} = a_{ij}x_{il} + b_{kl}$. Hence the derivative of $y_{kl}$ with respect to $x_{ij}$ is $a_{ij}$ when $l = j$, otherwise, the derivative will be 0.

- Take (2) into (1), we will have $\frac{\partial f}{\partial x_{ij}} = \sum_{kl} \frac{\partial f}{\partial y_{kl}} \frac{\partial y_{kl}}{\partial x_{ij}} = \sum_{kl} \frac{\partial f}{\partial y_{kl}} a_{kl} \delta_{ij} = \sum_{k} \frac{\partial f}{\partial y_{kl}} a_{k}$ (because only $y_{k}$ will be kept). In this equation, we know that $a_{k}$ is the $k$th row of $A^{T}$ and $\frac{\partial f}{\partial y_{k}}$ 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 $i$th row of $A^{T}$ and the $j$th column of $\nabla_{Y}f$.

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

$$\frac{\partial f}{\partial X} = \left[ \frac{\partial f}{\partial x_{11}} \quad \cdots \quad \frac{\partial f}{\partial x_{1n}} \right] \quad \cdots \quad \left[ \frac{\partial f}{\partial x_{p1}} \quad \cdots \quad \frac{\partial f}{\partial x_{pn}} \right] = A^{T} \nabla_{Y}f$$

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 $g(X) : \mathbb{R}^{m \times p} \rightarrow \mathbb{R}^{m \times n}, Y = g(X) = XC + D$ where $C \in \mathbb{R}^{p \times n}$ and $D \in \mathbb{R}^{m \times n}$. In this case, we have $f(Y) : \mathbb{R}^{m \times n} \rightarrow \mathbb{R}$ defined as well. Then to compute $\frac{\partial f}{\partial X}$, we first consider $Y^{T}$ and we have $Y^{T} = (XC + D)^{T} = CT^{T}X^{T} + DT^{T}$. 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 transformation 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 multiplication), 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 multiplication), 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.

### 4.2 Fully Connected Layers

**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 the below figure 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.

**Fig. 1: An Example of Fully Connected Layer**

From the definition of fully connected layer, we know that in the \( i \)th 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, \( m(1) = 2 \) (the blue nodes), \( m(2) = 2 \) (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 \( m(i) \times d \) 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 the example, we can compute \( h_1(1) \) as \( h_1(1) = x_1 w_1(1) + x_2 w_2(1) + x_0 \) where \( w_{jk} \) is the relation between \( x_j \) and \( h_k(1) \). 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 \( nd \) 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 \( \frac{\partial L}{\partial w} \) and \( \frac{\partial L}{\partial b} \). In the backward pass, the \( i \)th 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 \(i\)th layer, we have \( \frac{\partial L}{\partial b} \) known indeed.

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

- \( \frac{\partial L}{\partial w} = X^T \nabla^{(i)} \) (Using the Law 1)
With these results, we can update the weight and bias in the \(i\)th layer with \(w^{new} = w^{old} - \epsilon \nabla^{(i)} x^T\) and \(b^{new} = b^{old} - \epsilon \nabla^{(i)}\) where \(\epsilon\) 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 \(\frac{\partial L}{\partial x} = \nabla^{(i)} w^T\) (Using the Law 2).

### 4.2.1 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 below:

![Figure 2: Another Example of Fully Connected Layer](image)

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 \(2 \times 2\) matrix. We assume it to be \(w^{(0)} = \begin{bmatrix} w_{11}^{(0)} & w_{21}^{(0)} \\ w_{12}^{(0)} & w_{22}^{(0)} \end{bmatrix} = \begin{bmatrix} 0.15 & 0.20 \\ 0.25 & 0.30 \end{bmatrix}\) where \(w_{ki}^{(i)}\) is the weight between \(x_k^{(i)}\) and \(y_i^{(i)}\). For example, \(w_{11}^{(0)}\) is the weight between \(x_1^{(0)}\) and \(y_1^{(0)}\) 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, \(\hat{Y} = Xw^{(0)} + b^{(0)} = \begin{bmatrix} 0.15 \\ 0.25 \end{bmatrix} \begin{bmatrix} 0.15 & 0.20 \\ 0.25 & 0.30 \end{bmatrix} + [0.35, 0.45] = [0.49, 0.62]\)

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 \(\frac{\partial L}{\partial y_1} = \hat{y}_1 - y_1 = 0.48, \frac{\partial L}{\partial y_2} = \hat{y}_2 - y_2 = -0.37\). 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

- \(\frac{\partial L}{\partial w} = X^T\nabla^{(1)} = [0.1, 0.5]^T [0.48, -0.49] = \begin{bmatrix} 0.048 & -0.037 \\ 0.24 & -0.185 \end{bmatrix}\)
- \(\frac{\partial L}{\partial b} = \nabla^{(1)} = [0.48, -0.37]\)
- \(\frac{\partial L}{\partial x} = \nabla^{(1)} w^T = [0.48, -0.37] \begin{bmatrix} 0.15 & 0.20 \\ 0.25 & 0.30 \end{bmatrix} = [-0.002, 0.009]\)

### 4.2. Fully Connected Layers
The implementation of fully connected layer in tinyml is as below:

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:

Similarly, we can do the same procedures for $w^{(0)}_{12}$, $w^{(0)}_{21}$ and $w^{(0)}_{22}$. After computation, we have $\frac{\partial \ell}{\partial w^{(0)}_{12}} = -0.037$, $\frac{\partial \ell}{\partial w^{(0)}_{21}} = 0.1860625 - 0.1812625 = 0.02$ and $\frac{\partial \ell}{\partial w^{(0)}_{22}} = 0.184225 - 0.187925 = -0.0185$.

With these four values we can form the matrix of the loss value with respect to the weight matrix as $\begin{bmatrix} 0.048 & -0.037 \\ 0.24 & -0.185 \end{bmatrix}$ 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:

1. There are two biases in our case, $b^{(0)}_1 = 0.35$ and $b^{(0)}_2 = 0.45$. For $b^{(0)}_1$, we define $b_{left} = b^{(0)}_1 - \epsilon = 0.34$ and $b_{right} = b^{(0)}_1 + \epsilon = 0.36$. Then we have
   
   $$y_{left} = Xw + b^{(0)}_{left} = [0.1, 0.5] \begin{bmatrix} 0.15 \\ 0.25 \end{bmatrix} + [0.34, 0.45] = [0.48, 0.62],$$  
   and $\ell_{left} = \frac{1}{2}[(0.48 - 0.01)^2 + (0.62 - 0.99)^2] = 0.1789$.

   $$y_{right} = Xw + b^{(0)}_{right} = [0.1, 0.5] \begin{bmatrix} 0.15 \\ 0.25 \end{bmatrix} + [0.36, 0.45] = [0.50, 0.62],$$  
   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 $b^{(0)}_{11}$ as $\frac{\partial \ell}{\partial b^{(0)}_{11}} = \frac{\ell_{right} - \ell_{left}}{2\epsilon} = \frac{0.1885 - 0.1789}{0.02} = 0.48$.

2. For $b^{(0)}_{12}$, after computation, we have $\frac{\partial \ell}{\partial b^{(0)}_{12}} = \frac{0.18485 - 0.19225}{0.02} = -0.37$. 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:

1. There are two inputs in our example, the $x^{(0)}_1 = 0.1$ and $x^{(0)}_2 = 0.5$. For $x^{(0)}_1$, we define $x_{left} = x^{(0)}_1 - \epsilon = 0.09$, $x_{right} = x^{(0)}_1 + \epsilon = 0.11$ and $X_{left}, X_{right}$ are the corresponding matrices. Then we have:

   $$y_{left} = X_{left}w + b^{(0)} = [0.09, 0.5] \begin{bmatrix} 0.15 \\ 0.25 \end{bmatrix} + [0.35, 0.45] = [0.4885, 0.618],$$  
   and $\ell_{left} = \frac{1}{2}[(0.48 - 0.01)^2 + (0.62 - 0.99)^2] = 0.183673125$.

   $$y_{right} = X_{right}w + b^{(0)} = [0.11, 0.5] \begin{bmatrix} 0.15 \\ 0.25 \end{bmatrix} + [0.35, 0.45] = [0.4915, 0.622],$$  
   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 $x^{(0)}_1$ as $\frac{\partial \ell}{\partial x^{(0)}_1} = \frac{\ell_{right} - \ell_{left}}{2\epsilon} = \frac{0.183633125 - 0.183673125}{0.02} = -0.002$.

2. Similarly, for $x^{(0)}_2$, after computation, we have $\frac{\partial \ell}{\partial x^{(0)}_2} = \frac{0.183747625 - 0.183567625}{0.02} = 0.009$. 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.

The implementation of fully connected layer in tinyml is as below:
from tinyml.core import Backend as np
from .base import Layer

class Linear(Layer):
    '''
    Linear layer performs fully connected operation.
    '''
    def __init__(self, name, input_dim, output_dim):
        super().__init__(name)
        # Xavier Initialization
        weight = np.random.uniform(low=-np.sqrt(1 / input_dim),
                                     high=np.sqrt(1 / input_dim),
                                     size=(output_dim, input_dim))
        bias = np.random.uniform(low=-np.sqrt(1 / input_dim),
                                  high=np.sqrt(1 / input_dim),
                                  size=(output_dim))
        self.type = 'Linear'
        self.weight = self.build_param(weight)
        self.bias = self.build_param(bias)

def forward(self, input):
    '''
    The forward pass of fully connected layer is given by :math:`f(x)=wx+b`.
    '''
    # save input as the input will be used in backward pass
    self.input = input
    return np.matmul(input, self.weight.tensor.T) + self.bias.tensor

def backward(self, in_gradient):
    '''
    In the backward pass, we compute the gradient with respect to :math:`w`, :math:`b`, and :math:`x`.
    We have:
    .. math::
    \frac{\partial l}{\partial w} = \frac{\partial l}{\partial y}\frac{\partial y}{\partial w} = \frac{\partial l}{\partial y} x
    \frac{\partial l}{\partial b} = \frac{\partial l}{\partial y}
    \frac{\partial l}{\partial x_i} = \frac{\partial l}{\partial y}\frac{\partial y}{\partial x_i}
    '''
    self.weight.gradient = np.matmul(self.input.T, in_gradient).T
    self.bias.gradient = np.sum(in_gradient, axis=0)
    return np.matmul(in_gradient, self.weight.tensor)

4.3 ReLu

The purpose of using activation functions is to bring some non-linearity into the deep neural networks, so that the networks can fit the real world. One of the most popular activation function is the rectifier linear unit (ReLu).

The function is defined as $f(x) = max(0, x)$. Thus the forward pass is simple: $y_i = max(0, x_i)$.

In the ReLu function, we do not have any weight or bias to update. Hence we only need to compute the gradient to previous layer. We have $\frac{\partial l}{\partial x_i} = \frac{\partial l}{\partial y}\frac{\partial y}{\partial x_i}$. 

4.3. ReLu
Then we have:

\[
\frac{\partial l}{\partial x_i} = \begin{cases} 
0 & x_i < 0 \\
\frac{\partial l}{\partial y_i} & x_i > 0 \\
\text{undefined} & x_i = 0
\end{cases}
\]

We see that the derivative is not defined at the point \( x_i = 0 \), but when computing, we can set it to be 0, or 1, or any other values between.

The implementation of ReLu layer in tinyml is as below:

```python
from tinyml.core import Backend as np
from .base import Layer

class ReLu(Layer):
    '''
    ReLu layer performs rectifier linear unit operation.
    '''
    def __init__(self, name):
        super().__init__(name)
        self.type = 'ReLu'

    def forward(self, input):
        '''
        In the forward pass, the output is defined as :math:`y = \max(0, x)`.
        '''
        self.input = input
        return input * (input > 0)

    def backward(self, in_gradient):
        return in_gradient * (self.input > 0)
```

4.4 Dropout Layer

In deep neural networks, we may encounter over fitting when our network is complex and with many parameters. In Dropout: A Simple Way to Prevent Neural Networks from Overfitting, N.Srivastava et al proposed a simple technique named Dropout that could prevent overfitting. It refers to dropping out some neurons in a neural network randomly. The mechanism is equivalent to training different neural networks with different architecture in every batch.

The parameters for this layer is a preset probability \( p_0 \). It indicates the probability of dropping a neuron. For example, if \( p_0 = 0.5 \), then it means that every neuron in this layer has a 0.5 chance of being dropped. With the given probability, we can define the dropout layer to be a function \( y_i = f(x_i) \) such that

\[
y_i = \begin{cases} 
0 & r_i < p \\
x_i & r_i \geq p
\end{cases}
\]

, where \( r_i \) is randomly generated. However, if we use this function, the expectations of the output of dropout layer will be scaled to \( p_0 \). For example, if the original output is 1 and \( p_0 = 0.5 \), the output will become 0.5. This is unsatisfactory because when we are testing the neural networks, we do not want the output to be scaled. Thus, in practice we define the function to be

\[
y_i = \begin{cases} 
0 & r_i < p \\
x_i/p & r_i \geq p
\end{cases}
\]
Then the backward computation becomes straightforward:

\[
\frac{\partial l}{\partial x_i} = \begin{cases} 
0 & r_i < p \\
0 \times \frac{\partial l}{\partial y_i} \times \frac{\partial y_i}{\partial x_i} = \frac{1}{p} \frac{\partial l}{\partial y_i} & r_i \geq p
\end{cases}
\]

The implementation of dropout layer in tinyml is as below:

```python
from tinyml.core import Backend as np
from .base import Layer

class Dropout(Layer):
    '''
    Dropout Layer randomly drop several nodes.
    '''
    def __init__(self, name, probability):
        super().__init__(name)
        self.probability = probability
        self.type = 'Dropout'

    def forward(self, input):
        self.mask = np.random.binomial(1, self.probability, size=input.shape) / self.probability
        return (input * self.mask).reshape(input.shape)

    def backward(self, in_gradient):
        return in_gradient * self.mask
```

### 4.5 Softmax

Softmax is such a function that takes the output of the fully connected layers, and turn them into the probability. Formally, it takes an \(n\)-d vector, and normalizes it to \(n\) probabilities proportional to the exponentials of the input number. It is defined as

\[
f(x) = \frac{e^{x_i}}{\sum e^{x_j}}
\]

, where \(x_i\) is the \(i\)-th input number.

We can then compute the derivative by using the quotient rule (if \(f(x) = \frac{g(x)}{h(x)}\), then \(f'(x) = \frac{g'(x)h(x) - g(x)h'(x)}{h^2(x)}\)). In our case, we have \(g_i = e^{x_i}\) and \(h_i = \sum e^{x_j}\). Then we have \(\frac{\partial h_i}{\partial x_j} = e^{x_i} (i = j)\) or \(0 (i \neq j)\). For \(h_i\), no matter the relation between \(i\) and \(j\), the derivative will always be \(e^{x_i}\).

Thus we have:

When \(i = j\),

\[
\frac{\partial f}{\partial x_i} = \frac{e^{x_i} \sum e^{x_j} - e^{x_i} e^{x_j}}{(\sum e^{x_j})^2} = \frac{e^{x_i}}{\sum e^{x_j}} \times \frac{(\sum e^{x_j} - e^{x_i})}{\sum e^{x_j}} = f(x_i)(1 - f(x_i))
\]

When \(i \neq j\),

\[
\frac{\partial f}{\partial x_i} = \frac{0 - e^{x_i} e^{x_j}}{(\sum e^{x_j})^2} = -\frac{e^{x_i}}{\sum e^{x_j}} \times \frac{e^{x_j}}{\sum e^{x_j}} = -f(x_i)f(x_j)
\]

The implementation of softmax layer in tinyml is as below:
from tinyml.core import Backend as np

from .base import Layer

class Softmax(Layer):
    '''
    Softmax layer returns the probability proportional to the exponentials of the input number.
    '''
    def __init__(self, name='softmax', axis=1, eps=1e-10):
        super().__init__(name)
        self.type = 'Softmax'
        self.axis = 1
        self.eps = eps

    def forward(self, input):
        '''
        Some computational stability tricks here. > TODO: to add the tricks
        '''
        self.input = input
        shifted = np.exp(input - input.max(axis=1, keepdims=True))
        result = shifted / shifted.sum(axis=1, keepdims=True)
        return result

    def backward(self, in_gradient):
        '''
        Important: The actual backward gradient is not :math:`1`.
        The reason why we pass the gradient directly to previous layer is:
        since we know the formula is pretty straightforward when softmax is being used together
        with cross entropy loss (see theoretical induction), we compute the gradient in the
        cross entropy loss function, so that we could reduce the complexity, and increase
        the computational stabilities.
        '''
        return in_gradient

4.6 Max Pooling Layer

Pooling layer is another important component of convolutional neural networks. There are many ways of performing pooling in this layer, such as max pooling, average pooling, etc. In this part, we will only discuss max-pooling layer as it is used most commonly in convolutional neural networks.

In Max pooling layer, we also have a spatially small sliding window called the kernel. In the window, only the largest value will be remained and all other values will be dropped. For example, assume we have

\[ A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \]

and a $2 \times 2$ max-pooling kernel. Then the output $C$ will be

\[ C = \begin{bmatrix} 5 & 6 \\ 8 & 9 \end{bmatrix} \]
With the given kernel size $K_w$ and $K_h$, we can formalize the max-pooling process as

$$f(x_{ij}) = \begin{cases} 
  x_{ij} & x_{ij} \geq x_{mn}, \forall m \in [i-K_w, i+K_w], n \in [j-K_h, j+K_h] \\
  0 & \text{otherwise}
\end{cases}$$

Hence we can compute the derivative as below:

$$\frac{\partial l}{\partial x_{ij}} = \frac{\partial l}{\partial f} \frac{\partial f}{\partial x_{ij}} = \begin{cases} 
  \frac{\partial l}{\partial f} & x_{ij} \geq x_{mn}, \forall m \in [i-K_w, i+K_w], n \in [j-K_h, j+K_h] \\
  0 & \text{otherwise}
\end{cases}$$

The implementation of max pooling layer in tinyml is as below:

```python
from tinyml.core import Backend as np
from .base import Layer
from .convolution import col2im_indices, im2col_indices

class MaxPool2D(Layer):
    '''
    Perform Max pooling, i.e. select the max item in a sliding window.
    '''
    def __init__(self, name, input_dim, size, stride, return_index=False):
        super().__init__(name)
        self.type = 'MaxPool2D'
        self.input_channel, self.input_height, self.input_width = input_dim
        self.size = size
        self.stride = stride
        self.return_index = return_index
        self.out_height = (self.input_height - size[0]) / stride + 1
        self.out_width = (self.input_width - size[1]) / stride + 1
        if not self.out_height.is_integer() or not self.out_width.is_integer():
            raise Exception('Invalid dimension settings!')
        self.out_width = int(self.out_width)
        self.out_height = int(self.out_height)
        self.out_dim = (self.input_channel, self.out_height, self.out_width)

    def forward(self, input):
        self.num_of_entries = input.shape[0]
        input_reshaped = input.reshape(self.num_of_entries * input.shape[1], 1, input.shape[2], input.shape[3])
        self.input_col = im2col_indices(input_reshaped, self.size[0], self.size[1], padding=0, stride=self.stride)
        self.max_indices = np.argmax(self.input_col, axis=0)
        self.total_count = list(range(0, self.max_indices.size))
        output = self.input_col[self.max_indices, self.total_count]
        output = output.reshape(self.out_height, self.out_width, self.num_of_entries, self.input_channel).transpose(2, 3, 0, 1)
        indices = self.max_indices.reshape(self.out_height, self.out_width, self.num_of_entries, self.input_channel).transpose(2, 3, 0, 1)
        if self.return_index:
            return output, indices
```

(continues on next page)
4.7 Max Unpooling Layer

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.

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 \( \begin{bmatrix} 0 & 0 & 0 \\ 0 & 5 & 6 \\ 0 & 8 & 9 \end{bmatrix} \).
N, C, H, W = x_shape
H_padded, W_padded = H + 2 * padding, W + 2 * padding
x_padded = np.zeros((N, C, H_padded, W_padded), dtype=cols.dtype)
k, i, j = get_im2col_indices(x_shape, field_height, field_width, padding, stride)
cols_reshaped = cols.reshape(C * field_height * field_width, -1, N)
cols_reshaped = cols_reshaped.transpose(2, 0, 1)
x_padded[:, k, i, j] = cols_reshaped
if padding == 0:
    return
return x_padded[:, :, padding:-padding, padding:-padding]

class MaxUnpool2D(Layer):
    def __init__(self, name, input_dim, size, stride):
        super().__init__(name)
        self.type = 'MaxUnpool2D'
        self.input_channel, self.input_height, self.input_width = input_dim
        self.size = size
        self.stride = stride
        self.out_height = (self.input_height - 1) * stride + size[0]
        self.out_width = (self.input_width - 1) * stride + size[1]
        # it is definitely integer, so we do not need to check it anymore
        self.out_dim = (self.input_channel, self.out_height, self.out_width)

    def forward(self, input, max_indices):
        self.num_of_entries = input.shape[0]
        output_shape = (self.num_of_entries, self.out_dim[0], self.out_dim[1],
                        self.out_dim[2])
        indices = max_indices.reshape(input.shape)
        unpooled = np.zeros(output_shape)
        for i in range(self.num_of_entries):
            for j in range(self.input_channel):
                for m in range(self.input_height):
                    for n in range(self.input_width):
                        index = indices[i, j, m, n]
                        w_index = index % self.size[0]
                        h_index = index // self.size[1]
                        unpooled[i, j, m * self.stride + h_index,
                                 n * self.stride + w_index] = input[i, j, m, n]
        return unpooled

    def backward(self, in_gradient):
        return

class MaxUnpool2D(Layer):
    def __init__(self, name, input_dim, size, stride):
        super().__init__(name)
        self.type = 'MaxUnpool2D'
        self.input_channel, self.input_height, self.input_width = input_dim
        self.size = size
        self.stride = stride
        self.out_height = (self.input_height - 1) * stride + size[0]
        self.out_width = (self.input_width - 1) * stride + size[1]
        # it is definitely integer, so we do not need to check it anymore
        self.out_dim = (self.input_channel, self.out_height, self.out_width)

        def forward(self, input, max_indices):
            self.num_of_entries = input.shape[0]
            output_shape = (self.num_of_entries, self.out_dim[0], self.out_dim[1],
                            self.out_dim[2])
            indices = max_indices.reshape(input.shape)
            unpooled = np.zeros(output_shape)
            for i in range(self.num_of_entries):
                for j in range(self.input_channel):
                    for m in range(self.input_height):
                        for n in range(self.input_width):
                            index = indices[i, j, m, n]
                            w_index = index % self.size[0]
                            h_index = index // self.size[1]
                            unpooled[i, j, m * self.stride + h_index,
                                     n * self.stride + w_index] = input[i, j, m, n]
            return unpooled

        def backward(self, in_gradient):
            return

        def __call__(self, input, max_indices):
            return self.forward(input, max_indices)

4.7. Max Unpooling Layer 19
4.8 Convolutional Layer

**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 $A_{3 \times 3}$ and single filter matrix $B_{2 \times 2}$, and

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

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 $2 \times 2$ 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 seven loops.

Though the convolution operation can be computed by the above algorithm, we can still use matrix multiplication to perform such computation as suggested by *A guide to convolution arithmetic for deep learning*. 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 elements in the filters. For example, We can unroll the input and output in our case as $X_{3\times1} = [a_{11}, a_{12}, a_{13}, a_{21}, a_{22}, a_{23}, a_{31}, a_{32}, a_{33}]^T$ and $Y_{4\times1} = [y_{11}, y_{12}, y_{21}, y_{22}]^T$. Then we will want a $4 \times 9$ 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

$$W^*X^* = \begin{bmatrix} b_{11} & b_{12} & 0 & b_{21} & b_{22} & 0 & 0 & 0 & 0 \\ 0 & b_{11} & b_{12} & 0 & b_{21} & b_{22} & 0 & 0 & 0 \\ 0 & 0 & 0 & b_{11} & b_{12} & 0 & b_{21} & b_{22} & 0 \\ 0 & 0 & 0 & 0 & b_{11} & b_{12} & 0 & b_{21} & b_{22} \end{bmatrix} \times [a_{11}, a_{12}, a_{13}, a_{21}, a_{22}, a_{23}, a_{31}, a_{32}, a_{33}]^T = Y^*$$

**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 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).

### 4.8.1 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

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

### 4.8. Convolutional Layer

21
**Matrix Multiplication** 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.

The implementation of convolutional layer in tinyml is as below:

```python
from tinyml.core import GPU
from tinyml.core import Backend as np
from .base import Layer

def get_im2col_indices(x_shape, field_height=3, field_width=3, padding=1, stride=1):
    # First figure out what the size of the output should be
    N, C, H, W = x_shape
    # check if the output shape are integers
    assert (H + 2 * padding - field_height) % stride == 0
    assert (W + 2 * padding - field_height) % stride == 0
    out_height = (H + 2 * padding - field_height) / stride + 1
    out_width = (W + 2 * padding - field_width) / stride + 1
    # np.arange generates an evenly paced array, here it will be [0,1,2,..., field_height]
    i0 = np.repeat(np.arange(field_height, dtype='int32'), field_width)
    i0 = np.tile(i0, C)
    # Similarly, i1 will be [0,...,0, 1*stride,...,1*stride,..., out_width * stride]
    i1 = stride * np.tile(np.arange(out_width, dtype='int32'), int(out_height))
    i1 = i1.reshape(-1, 1) + j1.reshape(1, -1)
    k = np.repeat(np.arange(C, dtype='int32'), field_height * field_width).reshape(-1, 1)
    # shape of k: (C * field_height * field_width, 1)
```

(continues on next page)
# shape of j: (C * field_height * field_width, out_height * out_width)

# k for indicating channels

return (k, i, j)


def im2col_indices(x, field_height=3, field_width=3, padding=1, stride=1):
    
    # Zero-pad the input
    p = padding
    # add paddings to neighbors
    x_padded = np.pad(x, ((0, 0), (0, 0), (p, p), (p, p)), mode='constant')

    k, i, j = get_im2col_indices(x.shape, field_height, field_width, padding, stride)

    # get the columns with fancy indexing
    cols = x_padded[:, k, i, j]
    C = x.shape[1]
    cols = cols.reshape(1, 2, 0).reshape(field_height * field_width * C, -1)
    return cols


def col2im_indices(cols, x_shape, field_height=3, field_width=3, padding=1, stride=1):

    # An implementation of col2im based on fancy indexing and np.add.at

    N, C, H, W = x_shape
    H_padded, W_padded = H + 2 * padding, W + 2 * padding
    x_padded = np.zeros((N, C, H_padded, W_padded), dtype=cols.dtype)

    k, i, j = get_im2col_indices(x_shape, field_height, field_width, padding, stride)

    cols_reshaped = cols.reshape(C * field_height * field_width, -1, N)
    cols_reshaped = cols_reshaped.transpose(2, 0, 1)

    if GPU:
        # In cupy, scatter_add is equivalent to np.add.at
        np.scatter_add(x_padded, (slice(None), k, i, j), cols_reshaped)
    else:
        # ufunc.at performed unbuffered inplace operation.
        # (a, indices, b)
        # For addition ufunc, this method is equivalent to a[indices] += b,
        # except that results are accumulated for elements that are indexed
        # more than once
        np.add.at(x_padded, (slice(None), k, i, j), cols_reshaped)

    if padding == 0:
        return x_padded
    return x_padded[:, :, padding:-padding, padding:-padding]


class Conv2D(Layer):

    # Conv2D performs convolutional operation with given input.

```python
100 def __init__(self, name, input_dim, n_filter, h_filter, w_filter, stride, 
101 padding):
102     
103     :param input_dim: the input data dimension, it should be of the shape (N, C, H, W) where N is for number of input data, C is for the channel, H for height and W for width.
104     
105     :param n_filter: the number of filters used in this layer. It can be any integers.
106     
107     :param h_filter: the height of the filter.
108     
109     :param w_filter: the width of the filter.
110     
111     :param stride: the stride of the sliding filter.
112     
113     super().__init__(name)
114     self.type = 'Conv2D'
115     self.input_channel, self.input_height, self.input_width = input_dim
116     self.n_filter = n_filter
117     self.h_filter = h_filter
118     self.w_filter = w_filter
119     self.stride = stride
120     self.padding = padding
121     weight = np.random.randn( 
122     self.n_filter, self.input_channel, self.h_filter, 
123     self.w_filter) * np.sqrt( 
124     1.0 / (self.input_channel * self.h_filter * self.w_filter))
125     bias = np.random.randn(self.n_filter) * np.sqrt( 
126     1.0 / (self.input_channel * self.h_filter * self.w_filter))
127     self.weight = self.build_param(weight)
128     self.bias = self.build_param(bias)
129     self.out_height = (self.input_height - self.h_filter + 
130     2 * padding) / self.stride + 1
131     self.out_width = (self.input_width - self.w_filter + 
132     2 * padding) / self.stride + 1
133     if not self.out_width.is_integer() or not self.out_height.is_integer():
134         raise Exception("[tinyml] Invalid dimension settings!")
135     self.out_height, self.out_width = int(self.out_height), int(
136     self.out_width)
137     self.out_dim = (self.n_filter, self.out_height, self.out_width)
138
139 def forward(self, input):
140     self.n_input = input.shape[0]
141     self.input_col = im2col_indices(input, 
142     self.h_filter, 
143     self.w_filter, 
144     stride=self.stride, 
145     padding=self.padding)
146     weight_in_row = self.weight.tensor.reshape(self.n_filter, -1)
147     output = np.matmul(weight_in_row, 
148     self.input_col) + self.bias.tensor.reshape(
149     self.n_filter, 1)
150     output = output.reshape(self.n_filter, self.out_height, self.out_width, 
151     self.n_input)
152     output = output.transpose(3, 0, 1, 2)
153     return output
```
(continues on next page)
def backward(self, in_gradient):
    gradient_flat = in_gradient.transpose(1, 2, 3, 0).reshape(self.n_filter, -1)
    weight_gradient = np.matmul(gradient_flat, self.input_col.T)
    self.weight.gradient = weight_gradient.reshape(self.weight.tensor.shape)
    self.bias.gradient = np.sum(in_gradient, axis=(0, 2, 3)).reshape(self.n_filter)
    weight_flat = self.weight.tensor.reshape(self.n_filter, -1)
    out_gradient_col = np.matmul(weight_flat.T, gradient_flat)
    shape = (self.n_input, self.input_channel, self.input_height, self.input_width)
    out_gradient = col2im_indices(out_gradient_col, shape, self.h_filter, self.w_filter, self.padding, self.stride)
    return out_gradient

4.9 Transposed Convolutional Layer

Definition In Section 3.1.2 Convolution, we unrolled the filter from a $2 \times 2$ matrix into a $4 \times 9$ matrix, so that we can perform the convolution by matrix multiplication. After the convolution operation, the input data changes from a $3 \times 3$ matrix to a $2 \times 2$ matrix. The deconv operation is defined as the inverse of the convolution operation, i.e. change the input data from a $2 \times 2$ matrix into an output matrix with the shape $3 \times 3$ 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 $4 \times 9$ 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.

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 $3 \times 3$ 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 $3 \times 3$ 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 \([0, 255]\).

```python
from tinyml.core import Backend as np
from .base import Layer

def im2rows(input, inp_shape, filter_shape, dilation, stride, dilated_shape,
            padding, res_shape):
    ""
    Gradient transformation for the im2rows operation
    :param in_gradient: The grad from the next layer
    :param inp_shape: The shape of the input image
    :param filter_shape: The shape of the filter (num_filters, depth, height, width)
    :param dilation: The dilation for the filter
    :param stride: The stride for the filter
    :param dilated_shape: The dilated shape of the filter
    :param res_shape: The shape of the expected result
    :return: The reformed gradient of the shape of the image
    ""
    dilated_rows, dilated_cols = dilated_shape
    num_rows, num_cols = res_shape
    res = np.zeros(inp_shape, dtype=input.dtype)
    input = input.reshape((input.shape[0], input.shape[1], filter_shape[1],
                           filter_shape[2], filter_shape[3]))
    for it in range(num_rows * num_cols):
        # first found index of rows and columns
        # i for rows
        # j for columns
        i = it // num_rows
        j = it % num_rows
        # accessing via colons: [start:end:step]
        # commas are for different dimensions
        res[:, :, i * stride[0]:i * stride[0] + dilated_rows:dilation,
            j * stride[1]:j * stride[1] + dilated_cols:dilation] += input[:, it, :, :, :]
    if (padding != 0):
        # TODO: this only works for pad=1, right now.
        # remove the padding regions
        res = np.delete(res, 0, 2)
        res = np.delete(res, res.shape[2] - 1, 2)
        res = np.delete(res, 0, 3)
        res = np.delete(res, res.shape[3] - 1, 3)
    return res

class Deconv2D(Layer):
    ""
    Deconv2D performs deconvolution operation, or tranposed convolution.
    ""
    def __init__(self,
                 name,
                 input_dim,
                 n_filters,
                 h_filter,
                 w_filter,
                 stride,
                 (continues on next page)
```
```

dilation=1,
padding=0):

  ...
  :param input_dim: the input dimension, in the format of (C,H,W)
  :param n_filters: the number of convolution filters
  :param h_filter: the height of the filter
  :param w_filter: the width of the filter
  :param stride: the stride for forward convolution
  :param dilation: the dilation factor for the filters, =1 by default.
  ...
  super().__init__(name)
self.type = 'Deconv2D'
self.input_channel, self.input_height, self.input_width = input_dim
self.n_filters = n_filters
self.h_filter = h_filter
self.w_filter = w_filter
self.stride = stride
self.dilation = dilation
self.padding = padding
weight = np.random.randn(self.n_filters, self.input_channel,
                          self.h_filter, self.w_filter) / np.sqrt(
                          self.n_filters / 2.0)
bias = np.zeros((self.n_filters, 1))
self.weight = self.build_param(weight)
self.bias = self.build_param(bias)

def forward(self, input):
    filter_shape = self.weight.tensor.shape
    dilated_shape = ((filter_shape[2] - 1) * self.dilation + 1,
                      (filter_shape[3] - 1) * self.dilation + 1)
    res_shape = ((self.input_height - 1) * self.stride + dilated_shape[0],
                 (self.input_width - 1) * self.stride + dilated_shape[1])
    input_mat = input.reshape((input.shape[0], input.shape[1], -1)).transpose((0, 2, 1))
    filters_mat = self.weight.tensor.reshape(self.input_channel, -1)
    res_mat = np.matmul(input_mat, filters_mat)
    return im2rows(res_mat,
                   (input.shape[0], filter_shape[1], res_shape[0], res_shape[1]),
                   filter_shape, self.dilation, (self.stride, self.stride),
                   dilated_shape, self.padding, input.shape[2:])

def backward(self, in_gradient):
  ...
  This function is not needed in computation, at least right now.
  ...
  return in_gradient
```

4.9. Transposed Convolutional Layer
4.10 Mean Square Loss

The mean square error is defined as $l = \frac{1}{n} \sum (y_i - \hat{y}_i)^2$. Since this is the last derivative we need to compute, we will only need to compute $\frac{\partial l}{\partial y_i}$. Let $g(y_i) = y_i - \hat{y}_i$, then $\frac{\partial g}{\partial y_i} = 1$.

$$\frac{\partial l}{\partial y_i} = \frac{\partial l}{\partial g} \times \frac{\partial g}{\partial y_i} = \frac{2}{n}(y_i - \hat{y}_i)$$

The implementation of mean square error loss in tinyml is as below:

```python
def mse_loss(predicted, ground_truth):
    '''Compute the mean square error loss.'''
    diff = predicted - ground_truth.reshape(predicted.shape)
    return (diff**2).mean(), 2 * diff / diff.shape[1]
```

4.11 Cross Entropy Loss

The cross-entropy loss is defined as $l = -\sum_i^n \hat{y}_i \log(p(y_i))$ where $p(y_i)$ is the probability of the output number, i.e. we usually use cross-entropy loss after a softmax layer. By this nature, we could actually compute the derivative of cross-entropy loss with respect to the original output $y_i$ rather than $p(y_i)$.

Then we have:

$$\frac{\partial l}{\partial y_i} = -\sum_j \hat{y}_j \frac{\partial \log(p(y_j))}{\partial y_i} = -\sum_j \hat{y}_j \frac{1}{p(y_j)} \frac{\partial p(y_j)}{\partial y_i}$$

Then as we know there will be a $k = i$ such that $\frac{\partial p(y_k)}{\partial y_i} = p(y_i)(1 - p(y_i))$, and for other $k \neq i$, we have $\frac{\partial p(y_k)}{\partial y_i} = -p(y_j)p(y_i)$.

Then we have:

$$-\sum_j \hat{y}_j \frac{1}{p(y_j)} \frac{\partial p(y_j)}{\partial y_i} = (-y_i)(1 - p(y_i)) - \sum_{j \neq i} \hat{y}_j \frac{1}{p(y_j)} p(y_j)$$

$$= -y_i + p(y_i)y_i + \sum_{j \neq i} \hat{y}_j p(y_i)$$

$$= -y_i + p(y_i) \sum_{j \neq i} y_j$$

$$= -y_i + p(y_i) \sum_j p(y_j)$$

$$= p(y_i) - y_i$$

The form is very elegant, and easy to compute. Therefore we usually hide the computational process of the derivative of softmax in the computation of cross entropy loss.

The implementation of cross entropy loss in tinyml is as below:

```python
from tinyml.core import Backend as np
from tinyml.layers import Softmax

def cross_entropy_with_softmax_loss(predicted, ground_truth):
    softmax = Softmax('softmax')
    output_probabilities = softmax(predicted)
    print(output_probabilities)
    loss = np.mean(-np.log(output_probabilities[1])
```
```python
np.arange(output_probabilities.shape[0], dtype=np.int8),
ground_truth] + 1e-20))

output_probabilities[np.arange(output_probabilities.shape[0]),
ground_truth] -= 1

gradient = output_probabilities / predicted.shape[0]

return loss, gradient
```

4.11. Cross Entropy Loss
In this chapter, we will present three examples:

- A manually set neural networks. We will compute the training process by hand, and compare it with the result from tinyml.
- The hand written digits recognition with both fully connected neural networks and convolutional neural networks in tinyml.
- The cat classifier built with tinyml.

### 5.1 Back Propagation

In this section, we will use a fully connected neural network shown as below.

![Diagram of a simple neural network](image)

Fig. 1: Illustration of the simple neural network we will be using.

We will use the following notation to help us demonstrate the process. In the following, by default we will have

- $x_i$ is the input of the neural network. There are two inputs in our examples which are $x_1 = 0.25, x_2 = 0.65$ and the bias $x_0 = 0.30$. 
• $h^{(j)}_i$ are the data in $i$-th node of $j$-th layer. Then the weights that $j$-th layer use to multiply with its input is denoted by $w_{ij}^q$ where $q$ is the $q$-th node of $j$-th layer, and $p$ is the node in $(j-1)$-th layer. For example, the weight between $x_1$ and $h^{(1)}_1$ are denoted by $w_{11}^1$. In addition, we use $b_j$ to denote the bias in $j$-th layer. We assume that we have $w_{11}^1 = 0.20$, $w_{12}^1 = 0.25$, $w_{21}^1 = 0.30$, $w_{22}^1 = 0.35$, $b_1 = 0.40$ and $w_{11}^2 = 0.50$, $w_{12}^2 = 0.55$, $w_{21}^2 = 0.60$, $w_{22}^2 = 0.65$. In our case, there is only one bias in the hidden layer $h^{(1)}_0$ and we set it to be $h^{(1)}_0 = 0.20$

• $y_i$ is the $i$-th predicted output, and $y_i$ is the corresponding ground truth. In our case, we have $\hat{y}_1 = 0.99$ and $\hat{y}_2 = 0.01$.

• In this example, we will use mean square error, i.e. $l = \frac{1}{2} \sum_{i=1}^{2}(y_i - \hat{y}_i)^2$. We will use stochastic gradient descent (SGD) to optimize our weights, and in our optimizer, the learning rate $\lambda$ is set to be 0.1. We will use ReLu activation layer after every layer.

With these parameters, we can perform the first forward pass:

$$h^{(1)}_1 = x_1 \ast w_{11}^1 + x_2 \ast w_{12}^1 + x_0 = 0.25 \ast 0.20 + 0.65 \ast 0.30 + 0.30 = 0.545 \text{ and }$$

$$h^{(2)}_2 = x_1 \ast w_{11}^2 + x_2 \ast w_{12}^2 + x_0 = 0.25 \ast 0.25 + 0.65 \ast 0.35 + 0.30 = 0.59.$$ 

Then after the relu activation these values remained since they are all positive, then

$$\hat{y}_1 = h^{(1)}_1 \ast h^{(1)}_1 + h^{(1)}_1 \ast h^{(1)}_1 + h_0 = 0.545 \ast 0.50 + 0.59 \ast 0.60 + 0.20 = 0.8265$$

$$\hat{y}_2 = h^{(1)}_1 \ast w_{12}^2 + h^{(1)}_1 \ast w_{22}^2 + h_0 = 0.545 \ast 0.55 + 0.59 \ast 0.65 + 0.20 = 0.88325.$$ 

Then after the relu activation these values remained since they are all positive.

Then we can compute the loss with this iteration, $l = \frac{1}{2} \sum_{i=1}^{2}(y_i - \hat{y}_i)^2 = \frac{1}{2}(0.8265 - 0.99)^2 + (0.88325 - 0.01)^2 = 0.39465$. Then the gradient will be

$$\frac{\partial l}{\partial y_1} = (y_1 - \hat{y}_1) = 0.8265 - 0.99 = -0.1635 \text{ and }$$

$$\frac{\partial l}{\partial y_2} = (y_2 - \hat{y}_2) = 0.88325 - 0.01 = 0.87325.$$ 

After this, we want to compute the derivative of the loss with regards to the weight and bias in the output layer. We have

$$\frac{\partial l}{\partial w_{11}^1} = \frac{\partial l}{\partial y_1} \frac{\partial y_1}{\partial w_{11}^1} = -0.1635 \ast h^{(1)}_1 = -0.1635 \ast 0.545 = -0.0891075$$

$$\frac{\partial l}{\partial w_{12}^1} = \frac{\partial l}{\partial y_1} \frac{\partial y_1}{\partial w_{12}^1} = 0.87325 \ast h^{(1)}_1 = 0.87325 \ast 0.545 = 0.47592125$$

$$\frac{\partial l}{\partial w_{21}^1} = \frac{\partial l}{\partial y_1} \frac{\partial y_1}{\partial w_{21}^1} = -0.1635 \ast h^{(1)}_1 = -0.1635 \ast 0.59 = -0.096465$$

$$\frac{\partial l}{\partial w_{22}^1} = \frac{\partial l}{\partial y_1} \frac{\partial y_1}{\partial w_{22}^1} = 0.87325 \ast h^{(1)}_1 = 0.87325 \ast 0.59 = 0.5152175.$$ 

Then we found that the bias $h_0$ will impact the loss in two ways: by impacting $y_1$ and $y_2$. Thus we can first convert the bias into a vector, and optimize the values in the vector accordingly. For example, in our case, we have

$$h^{(1)}_0 = [h^{(1)}_{01}, h^{(1)}_{02}] = [0.20, 0.20], \text{ and the gradient will be }$$

$$\frac{\partial l}{\partial h^{(1)}_0} = \frac{\partial l}{\partial y_1} \frac{\partial y_1}{\partial h^{(1)}_0} = -0.1635$$

32 Chapter 5. Examples
\[
\frac{\partial l}{\partial \omega_{12}} = \frac{\partial l}{\partial y_2} \frac{\partial y_2}{\partial \omega_{12}} = \frac{\partial l}{\partial y_2} = 0.87325
\]

Then the gradient that we will pass to the previous layer, i.e. \( \frac{\partial l}{\partial \omega_{ij}} \), can be computed as below:

\[
\frac{\partial l}{\partial \omega_{11}^{(1)}} = \frac{\partial l}{\partial y_1} \frac{\partial y_1}{\partial \omega_{11}^{(1)}} + \frac{\partial l}{\partial y_2} \frac{\partial y_2}{\partial \omega_{11}^{(1)}} = -0.1635 * w_{11}^2 + 0.87325 * w_{12}^2 = -0.1635 * 0.50 + 0.87325 * 0.55 = 0.3985375
\]

\[
\frac{\partial l}{\partial \omega_{21}^{(1)}} = \frac{\partial l}{\partial y_1} \frac{\partial y_1}{\partial \omega_{21}^{(1)}} + \frac{\partial l}{\partial y_2} \frac{\partial y_2}{\partial \omega_{21}^{(1)}} = -0.1635 * w_{21}^2 + 0.87325 * w_{22}^2 = -0.1635 * 0.60 + 0.87325 * 0.65 = 0.4695125
\]

With these derivatives, we can update the weight and bias in the output layer, here we will use the learning rate \( \lambda = 0.1 \). Then

\[
w_{11}^2 = w_{11}^2 - \lambda \frac{\partial l}{\partial w_{11}^{(1)}} = 0.50 - 0.1 * (-0.0891075) = 0.50891075
\]

\[
w_{12}^2 = w_{12}^2 - \lambda \frac{\partial l}{\partial w_{12}^{(1)}} = 0.55 - 0.1 * (0.47592125) = 0.502407875
\]

\[
w_{21}^2 = w_{21}^2 - \lambda \frac{\partial l}{\partial w_{21}^{(1)}} = 0.60 - 0.1 * (-0.096465) = 0.6096465
\]

\[
w_{22}^2 = w_{22}^2 - \lambda \frac{\partial l}{\partial w_{22}^{(1)}} = 0.65 - 0.1 * (0.5152175) = 0.59847825
\]

\[
h_{01}^2 = h_{01}^2 - \lambda \frac{\partial l}{\partial h_{01}^{(1)}} = 0.20 - 0.1 * (-0.1635) = 0.21635
\]

\[
h_{02}^2 = h_{02}^2 - \lambda \frac{\partial l}{\partial h_{02}^{(1)}} = 0.20 - 0.1 * (0.87325) = 0.112675
\]

Until now, we have successfully updated the value in the output layer, and afterwards, we will need to iteratively update the value in previous layers. Before working on the iterative process, in order to make the computation clean, we will use the matrix format. Here we already have the gradient that the output layer passed to hidden layer denoted by \( \nabla_{2} = [\frac{\partial l}{\partial \omega_{11}^{(1)}}, \frac{\partial l}{\partial \omega_{12}^{(1)}}] = [0.3985375, 0.4695125] \). Then

\[
\frac{\partial l}{\partial w_{ij}^{(1)}} = \frac{\partial l}{\partial \omega_{ij}^{(1)}}. \quad \text{Thus we will have}
\]

\[
\begin{bmatrix}
\frac{\partial l}{\partial w_{11}^{(1)}} \\
\frac{\partial l}{\partial w_{12}^{(1)}} \\
\frac{\partial l}{\partial w_{21}^{(1)}} \\
\frac{\partial l}{\partial w_{22}^{(1)}}
\end{bmatrix} =
\begin{bmatrix}
0.25 \\
0.65 \\
0.20 \\
0.35
\end{bmatrix}
\]

\[
= \begin{bmatrix}
0.3985375 & 0.4695125 \\
0.09963437 & 0.11737812 \\
0.25904938 & 0.30518312
\end{bmatrix}
\]

Then for the bias, we have \( \frac{\partial l}{\partial x_0} = \frac{\partial l}{\partial h_j} \). Therefore \( \frac{\partial l}{\partial x_0} = [0.3985375, 0.4695125] \).

With these parameters, we can update the parameters in the hidden layer.

\[
w_{11}^1 = w_{11}^1 - \lambda \frac{\partial l}{\partial w_{11}^{(1)}} = 0.20 - 0.1 * (0.09963437) = 0.190036563
\]

\[
w_{12}^1 = w_{12}^1 - \lambda \frac{\partial l}{\partial w_{12}^{(1)}} = 0.25 - 0.1 * (0.11737812) = 0.238262188
\]

\[
w_{21}^1 = w_{21}^1 - \lambda \frac{\partial l}{\partial w_{21}^{(1)}} = 0.30 - 0.1 * (0.25904938) = 0.274095062
\]

\[
w_{22}^1 = w_{22}^1 - \lambda \frac{\partial l}{\partial w_{22}^{(1)}} = 0.35 - 0.1 * (0.30518312) = 0.319481688
\]

\[
x_{01} = x_{01} - \lambda \frac{\partial l}{\partial x_{01}} = 0.20 - 0.1 * (0.3985375) = 0.16014625
\]

\[
x_{02} = x_{02} - \lambda \frac{\partial l}{\partial x_{02}} = 0.20 - 0.1 * (0.4695125) = 0.15304875
\]

Until now, we have successfully computed a forward pass, a backward pass and updated all the parameters in the neural network. With the same procedure, we can compute the new loss \( l = 0.2384542802724947 \). If the new loss is still unsatisfactory, we can repeat the process again and again to lower the loss until it is satisfactory.

The process can be easily implemented in tinyml with the following code snippets.

5.1. Back Propagation 33
import numpy as np
import tinyml
from tinyml.layers import Linear, ReLu
from tinyml.losses import mse_loss
from tinyml.net import Sequential
from tinyml.optims import SGD Optimizer

tinyml.utilities.logger.VERBOSE = 3

x = np.array([0.25, 0.65]).reshape(1, 2)
y = np.array([0.99, 0.01]).reshape(1, 2)
print(x.shape)
p
t = np.array([0.20, 0.25, 0.30, 0.35]).reshape(2, 2)
f1 = Linear('fc_1', 2, 2)
f2 = Linear('fc_2', 2, 2)
relu = ReLu('relu_1')
f1.weight = t
f1.bias = np.array([0.30, 0.30])
f1._rebuild_params()
f2.weight = np.array([0.50, 0.55, 0.60, 0.65]).reshape(2, 2)
f2.bias = np.array([0.20, 0.20])
f2._rebuild_params()

model = Sequential([f1, relu, f2])
optimizer = SGD Optimizer(0.1)
model.summary()

for epoch in range(epoch):
    y_predicted = model.forward(x)
    loss, loss_gradient = mse_loss(y_predicted, y)
    print('>>>>')
p
    loss
    print(loss)
    print(loss_gradient)
    print('<<<<')
    model.backward(loss_gradient)
    model.update(optimizer)
5.2 The Hand Written Digits Recognition

With fully connected neural networks, the task can be achieved by:

```python
import os

from sklearn.preprocessing import OneHotEncoder

import tinyml
import tinyml.dataloaders.mnist as mnist
from tinyml.core import Backend as np
from tinyml.layers import Conv2D, Dropout, Linear, ReLu, Softmax, softmax
from tinyml.layers.pooling import MaxPool2D
from tinyml.layers.flatten import Flatten

from tinyml.learner import Learner
from tinyml.learner.callbacks import evaluate_classification_accuracy
from tinyml.losses import cross_entropy_with_softmax_loss, mse_loss
from tinyml.net import Sequential

# Higher verbose level = more detailed logging
tinyml.utilities.logger.VERBOSE = 1

GPU = False

if GPU:
    os.environ['TNN_GPU'] = "True"

print('loading data...')
# mnist.init()

def pre_process_data(train_x, train_y, test_x, test_y):
    # Normalize
    train_x = train_x / 255.
    test_x = test_x / 255.

    return train_x, train_y, test_x, test_y

x_train, y_train, x_test, y_test = mnist.load()

x_train, y_train, x_test, y_test = pre_process_data(x_train, y_train, x_test, y_test)

if GPU:
    import cupy as cp
    x_train = cp.array(x_train)
    y_train = cp.array(y_train)
    x_test = cp.array(x_test)
    y_test = cp.array(y_test)

print(y_train.shape)
print(x_train.shape)
print('building model...')

model = Sequential(
    Linear('fc_1', 784, 128),
    ...
)
```

(continues on next page)
With convolutional neural networks, the task can be achieved by:

```python
import os
from sklearn.preprocessing import OneHotEncoder
import tinyml
import tinyml.dataloaders.mnist as mnist
from tinyml.core import Backend as np
from tinyml.layers import Conv2D, Dropout, Linear, ReLu, Softmax, softmax
from tinyml.layers.flatten import Flatten
from tinyml.layers.pooling import MaxPool2D
from tinyml.learner import Learner
from tinyml.learner.callbacks import evaluate_classification_accuracy
from tinyml.losses import cross_entropy_with_softmax_loss, mse_loss
from tinyml.net import Sequential
from tinyml.optims import SGDOptimizer

# Higher verbose level = more detailed logging
tinyml.utilities.logger.VERBOSE = 1

GPU = False

if GPU:
    os.environ['TNN_GPU'] = "True"

print('loading data...')
# mnist.init()

def pre_process_data(train_x, train_y, test_x, test_y):
    # Normalize
    (continued on next page)
train_x = train_x / 255.

test_x = test_x / 255.

train_x = train_x.reshape(-1, 1, 28, 28)
test_x = test_x.reshape(-1, 1, 28, 28)

return train_x, train_y, test_x, test_y

x_train, y_train, x_test, y_test = mnist.load()
x_train, y_train, x_test, y_test = pre_process_data(x_train, y_train, x_test, y_test)

if GPU:
    import cupy as cp
    x_train = cp.array(x_train)
y_train = cp.array(y_train)
x_test = cp.array(x_test)
y_test = cp.array(y_test)

print(y_train.shape)
print(x_train.shape)

print('building model...')

model = Sequential([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),
    n_filter=64,
    h_filter=3,
    w_filter=3,
    stride=1,
    padding=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)
])

model.build_params()

model.summary()
callbacks = [evaluate_classification_accuracy]
cargs = (x_test, y_test)
learner = Learner(model, cross_entropy_with_softmax_loss, SGDOptimizer(lr=0.2))

print('starting training...')
learner.fit(x_train, y_train, epochs=5,

(continues on next page)
5.3 The Cat Classifier

To be added soon.
6.1 Layers

class tinyml.layers.Conv2D (name, input_dim, n_filter, h_filter, w_filter, stride, padding)
Conv2D performs convolutional operation with given input.

    backward (in_gradient)
    > this function needs to be overridden.

    forward (input)
    > this function needs to be overridden. compute the forward pass

class tinyml.layers.Deconv2D (name, input_dim, n_filters, h_filter, w_filter, stride, dilation=1, padding=0)
Deconv2D performs deconvolution operation, or transposed convolution.

    backward (in_gradient)
    This function is not needed in computation, at least right now.

    forward (input)
    > this function needs to be overridden. compute the forward pass

class tinyml.layers.Dropout (name, probability)
Dropout Layer randomly drop several nodes.

    backward (in_gradient)
    > this function needs to be overridden.

    forward (input)
    > this function needs to be overridden. compute the forward pass

class tinyml.layers.Flatten (name)
Flatten layer reads an ndarray as input, and reshape it into a 1-d vector.

    backward (in_gradient)
    > this function needs to be overridden.

    forward (input)
    > this function needs to be overridden. compute the forward pass

class tinyml.layers.Linear (name, input_dim, output_dim)
Linear layer performs fully connected operation.

    backward (in_gradient)
    In the backward pass, we compute the gradient with respect to $w$, $b$, and $x$. 
We have:

\[
\frac{\partial l}{\partial w} = \frac{\partial l}{\partial y} \frac{\partial y}{\partial w} = \frac{\partial l}{\partial y} \frac{\partial y}{\partial x}
\]

**forward**(*input*)
The forward pass of fully connected layer is given by \( f(x) = wx + b \).

**class** tinyml.layers.MaxPool2D(*name*, *input_dim*, *size*, *stride*, *return_index=False*)
Perform Max pooling, i.e. select the max item in a sliding window.

**backward**(*in_gradient*)
> this function needs to be overridden.

**forward**(*input*)
> this function needs to be overridden. compute the forward pass

**class** tinyml.layers.ReLU(*name*)
ReLU layer performs rectifier linear unit operation.

**backward**(*in_gradient*)
> this function needs to be overridden.

**forward**(*input*)
In the forward pass, the output is defined as \( y = \max(0, x) \).

**class** tinyml.layers.Softmax(*name='softmax', axis=1, eps=1e-10*)
Softmax layer returns the probability proportional to the exponentials of the input number.

**backward**(*in_gradient*)
Important: The actual backward gradient is not 1.

The reason why we pass the gradient directly to previous layer is: since we know the formula is pretty straightforward when softmax is being used together with cross entropy loss (see theoretical induction), we compute the gradient in the cross entropy loss function, so that we could reduce the complexity, and increase the computational stabilities.

**forward**(*input*)
Some computational stability tricks here. > TODO: to add the tricks

### 6.2 Optimizers

**class** tinyml.optim.SGD**(Optimizer**(lr, momentum=\(None\))
In this class, we implement the stochastic gradient descent algorithm, which is used to update the parameters in a neural network. The algorithm is simple:

\[
\mathbf{w}_{\text{new}} = \mathbf{w}_{\text{old}} - \lambda \nabla
\]

where \(\lambda\) is the preset learning rate, and \(\nabla\) is the gradient.
6.3 Loss Functions

tinyml.losses.mse_loss(predicted, ground_truth)
Compute the mean square error loss.

6.4 Net

class tinyml.net.Sequential(layers)
Sequential model reads a list of layers and stack them to be a neural network.
INDICES AND TABLES

- genindex
- modindex
- search
t
  tinyml.layers, 39
  tinyml.losses, 41
  tinyml.net, 41
  tinyml.optims, 40
INDEX

B
backward() (tinyml.layers.Conv2D method), 39
backward() (tinyml.layers.Deconv2D method), 39
backward() (tinyml.layers.Dropout method), 39
backward() (tinyml.layers.Flatten method), 39
backward() (tinyml.layers.Linear method), 39
backward() (tinyml.layers.MaxPool2D method), 40
backward() (tinyml.layers.ReLU method), 40
backward() (tinyml.layers.Softmax method), 40

C
Conv2D (class in tinyml.layers), 39

D
Deconv2D (class in tinyml.layers), 39
Dropout (class in tinyml.layers), 39

F
Flatten (class in tinyml.layers), 39
forward() (tinyml.layers.Conv2D method), 39
forward() (tinyml.layers.Deconv2D method), 39
forward() (tinyml.layers.Dropout method), 39
forward() (tinyml.layers.Flatten method), 39
forward() (tinyml.layers.Linear method), 40
forward() (tinyml.layers.MaxPool2D method), 40
forward() (tinyml.layers.ReLU method), 40
forward() (tinyml.layers.Softmax method), 40

L
Linear (class in tinyml.layers), 39

M
MaxPool2D (class in tinyml.layers), 40
module
tinyml.layers, 39
tinyml.losses, 41
tinyml.net, 41
tinyml.optims, 40
mse_loss() (in module tinyml.losses), 41

R
ReLU (class in tinyml.layers), 40

S
Sequential (class in tinyml.net), 41
SGD Optimizer (class in tinyml.optims), 40
Softmax (class in tinyml.layers), 40

T
tinyml.layers module, 39
tinyml.losses module, 41
tinyml.net module, 41
tinyml.optims module, 40