Lab 4 - A Convolutional Neural Network from Scratch

ARIZONA STATE UNIVERSITY
SCHOOL OF ELECTRICAL, COMPUTER,
AND ENERGY ENGINEERING,
EEE598: Deep Learning Media Processing & Understanding

1 Objectives

- Understand and implement backwards and forwards passes of a convolutional layer
- Understand and implement backwards and forwards passes of a max pooling layer
- Understand and implement backwards and forwards passes of a flattening layer
- Test a convolutional neural network on a subset of the CIFAR100 dataset

2 Convolutional Layer

In this lab, we will be implementing convolutional layers and construct a basic convolutional neural network. Why do we need convolutional layers? Theoretically, the fully connected layers can model any function. In practice, learning the function with fully connected layers may be difficult because of the large number of parameters. For example if our input is a $244 \times 224 \times 3$ image, and our fully connected layer has a modest output size of 500, then we need $3 \times 224 \times 224 \times 500 =\sim 75$million weights for a single layer. This is clearly not desirable as our dataset probably has much fewer samples than this, so the model will likely have an overfitting problem. One solution to this problem is to utilize *weight sharing*, of which convolutional neural networks is one popular type.

Convolution operations are useful when we expect our data to exhibit translations in some dimensions. Using convolution to recognize entities in images is intuitive because we expect image data to have some degree of translational invariance. For example, if we are trying to detect a car, it would be useful to detect wheels. These wheels could be anywhere in the image, so we would like our wheel detector to be translationally invariant. Before the rise of deep learning, this was often accomplished by a "sliding-window" detector. The convolution operator is analogous to the sliding window approach. The difference is that convolution implements a simple multiply add operation instead of being a full classifier, as in the traditional sliding window approach. As in the fully connected networks, a stacked network of convolutional layers can learn complex functions of the input data.

The convolution operation uses a kernel, or filter, to compute outputs. In convolutional networks for image recognition, the filter size is usually small (e.g., 3×3 or 7×7). The weights of the filters are learnable parameters. Additionally, the convolutional layer usually incorporates a learnable bias term, which is added to the filter output. As the filter slides across the input, outputs are generated for each input location. At the borders of the image, input data for the convolution is missing. This data can be replaced with zeros (zero padding), or the convolution output at these locations can be ignored ("valid" convolution). In this lab we will consider convolutions that slide over 2 dimensions,

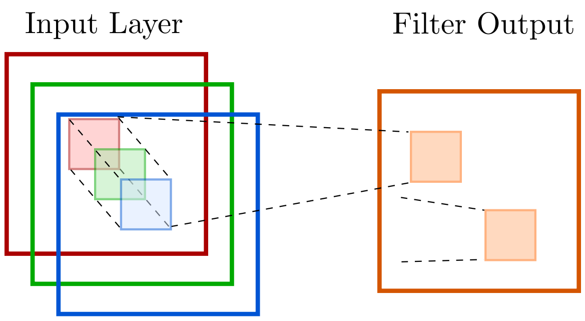


Figure 1: Example of convolutional layer operation for a single output map.

so the output of our filter will be a 2D map of the outputs corresponding to different locations. It is also possible to do convolutions in 1 dimension (e.g., for audio applications), or convolution in higher dimensions (e.g., applications using 3D point clouds, or video data).

We allow the convolutional layer to have multiple filters, which would give multiple outputs. Often the outputs of the layer are called channels, analogous to image channels because the output channels have the same spatial dimensions. Each filter performs a convolution over the spatial dimensions (row and column). The filter also operates over the channel dimension of the input, but the filter does not "slide" over this dimension. For a first convolutional layer in a network trained on image data, the filters will be RGB filters (assuming image data). For convolutional layers not connected to the input, the input channels represent some intermediate representation learned by the network. Each of the filters has different learnable parameters, but we will combine all of the filter parameters into a single 4 dimensional tensor for each particular layer. Each filter has a separate learnable bias term, so we store the biases in a vector. Figure 1 shows a visual example of the convolution operation.

The implementation of the convolutional layer will use the same class interface as we used in Lab 3. The convolutional layer will be implemented in `conv.py` and the tests are given in `test_conv.py`. You need to implement the `forward`, `backward`, `update_param`, and `__init__` methods.

Again, the convolutional layer will "learn" good filters to use for our particular problem. It is useful to visualize properties of our network so that we gain some intuition as to how the network is working. There are many ways to visualize properties of the network, but the simplest way is to visualize the filter weights. This is easiest to do at the input layer, because the filters can be visualized as color images. Figure 2 shows filters from the AlexNet model [1] trained on a large dataset. We can see that these learned filters look like different edge and blob detectors. In this case the filter size is $3 \times 11 \times 11$. 3 corresponds to the number of input channels, and 11×11 is often referred to as the filter size.

2.1 Data and Dimensions

The dimensions of our data are slightly complicated to keep track of, so it is important to be aware of them. If you perform an operation on the wrong dimension, the output will be incorrect. The order of the dimensions is arbitrary; in fact, even different machine learning libraries have different dimension ordering. To pass the tests, your code will need to follow the dimensions listed in this section.

For the variables corresponding to data, we will always leave the first dimension to correspond to individual data samples. The size of this first dimension will be the size of the batch, since we will process one batch at a time. The next dimension is the channel dimension. By "channels" here we mean the number of input/output filter responses in each layer. For example an input color image has 3 channels. A convolutional layer with 16 filters would result in an output with 16

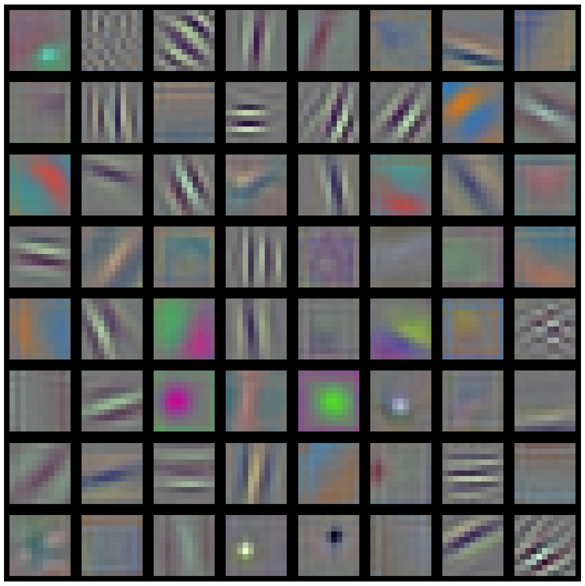


Figure 2: First layer filters from AlexNet. The first convolutional layer operates on the RGB input image, so we can visualize the filters as color images. We can see that the layer tends to learn Gabor-like filters

channels. Each channel can be thought of as a two dimensional map of responses. Finally, the last two dimensions are the rows and columns of this two dimensional map.

Table 1 lists the expected sizes of different variables and function outputs of the convolutional layer.

2.2 Forward pass

The data passed into a convolutional layer is multidimensional with dimensions (n_b, n_i, n_r, n_c), where n_b is the mini-batch size, n_i is the number of input channels (also called input depth), n_r is the number of rows of the input, and n_c is the number of columns of the input. The output will be of dimension (n_b, n_o, n_r, n_c), where n_o is the number of output filter maps.

If we have n_o filters of size $n_i \times h \times h$, we store the filter weights in a variable `W` with shape (n_o, n_i, h, h). The channel dimension of the filter is almost always equal to the number of input channels. An example of an exception to this case is AlexNet [1], where input channels are grouped to enable computation on 2 GPUs. Thus, the filter will only move in the two spatial dimensions. In addition, to the convolution weights it is also common to add a bias, as in the fully connected layer. There are two ways to implement the bias: *tied* bias and *untied* bias. For the untied bias, there is a different bias for each corresponding output location. For the tied bias, there is one bias corresponding to each output channel. In this case the bias is a vector of size n_o. In this lab, we will implement the tied bias configuration.

For a single output map (indexed by o), and a single input sample (indexed by i), the convolution layer implements the following:

$$f_{conv_o}(\mathbf{x}_i) = \mathbf{W}_o \overset{\text{corr}}{**} \mathbf{x}_i + \mathbf{b}_o \tag{1}$$

where $\mathbf{x}_i$ is a $(1, n_i, n_r, n_c)$ shaped input from the previous layer, $\mathbf{W}_o$ is a single filter corresponding to the oth output, $\mathbf{b}_o$ is a single bias corresponding to the output indexed by o, and $**$ is the convolution operator.

Table 1: Data dimensions

Variable	Description	1st dim	2nd dim	3rd dim	4th dim
`x`	Input	batch size	# of input channels	rows	columns
`forward(x)`	Output of forward pass	batch size	# of output channels	rows	columns
`y_grad`	Gradient at output	batch size	# of output channels	rows	columns
`backward(y_grad)`	Output of backward pass	batch size	# of input channels	rows	columns
`W`	Weight tensor	# of output channels	# of input channels	rows	columns
`W_grad`	Gradient of weight tensor	# of output channels	# of input channels	rows	columns
`b`	Bias vector	1	# of output channels	-	-
`b_grad`	Gradient of bias vector	1	# of output channels	-	-

One thing to be careful with convolution is the behavior at the edges of the input. What is the behavior of convolution if the receptive field overlaps with the border? In *valid* convolution, the outputs of the locations that overlap with the border are ignored. In *full* convolution, we pad the inputs with some value (typically 0) and then we are able to compute outputs corresponding to every input. In our implementation we will be using *full* convolution.

In signal processing, we are taught to flip the filter before we slide the filter across the input. Many machine learning libraries do not perform such flipping, because the flipping makes no difference when training a convolutional neural network. In our implementation, we will be doing convolution without flipping the filter to match the implementation in common machine learning libraries (e.g. PyTorch). We can use the `scipy.signal.correlate` to perform the convolution with flipped filters. If the `scipy.signal.convolve` is used instead, you need to first flip the filters to obtain the same result as `scipy.signal.correlate`.

We will illustrate the desired behavior of the forward pass of the convolutional network with a small example. The input to our layer will be of size $1, 2, 4, 4$. This would correspond to a batch with a single image with two channels, and spatial dimensions of 4×4. We will apply 2 filters of size $2, 3, 3$, so our weight tensor will be of size $2, 2, 3, 3$. The output will be of size $1, 2, 4, 4$. Figure 3 shows this convolution operation for a fake input without using a bias term.

We could implement the convolution ourself with loops. It is easier, however, to use a convolution function from one of the included Python libraries. In this case we need to do convolution with flipped filters, also called correlation. We can use the `scipy.signal.correlate` function. Our inputs and outputs have four dimensions (batch, channel, rows, columns), but we only want to do proper convolution for two of the dimensions (rows and columns). If we use the `scipy.signal.correlate` with "full" correlation naively, then we will not get the expected result, because the function will perform zero padding in all dimensions. The "valid" setting does not perform zero padding, but we actually do want zero padding for the row and column dimensions. So we can either zero pad the data in the appropriate dimensions ourselves (using the `np.pad` function), or we can use loops to perform separate convolutions.

Channel 0

0	0	0	0
1	-1	1	-1
2	-2	2	-2
3	-3	3	-3

3	2	1
3	2	1
3	2	1

3	3	3
2	2	2
1	1	1

1	2	-2	1
6	14	8	11
3	4	-26	-2
5	10	-10	5

Channel 1

0	1	2	3
0	-1	-2	-3
0	1	2	3
0	-1	-2	-3

1	2	3
1	2	3
1	2	3

1	1	1
2	2	2
3	3	3

-1	-2	-7	-5
2	10	8	10
-2	4	-22	-10
-1	9	-18	-5

(a) Input **(b)** Filter 0 **(c)** Filter 1 **(d)** Output

Figure 3: Convolutional forward pass example with no bias term. This example assumes the filters are already flipped.

2.3 Backwards Pass

For the backwards pass of the convolutional layer we need to compute $\frac{d\ell}{d\mathbf{W}}$, $\frac{d\ell}{d\mathbf{b}}$, and $\frac{d\ell}{d\mathbf{x}}$. Recall that ℓ is the loss function at the end of the network. To compute these gradients we assume that we have the gradient at the output of the considered layer l as $\frac{d\ell}{d\mathbf{x}^{l+1}}$. Similar to backpropagation from the previous lab, the `backward` function has an input `y_grad` that represents the gradient of the output of the neural network with respect to the output of the current layer.

First we will start with the easiest case: the gradient of `b`. This gradient will be stored in the class variable `b_grad`. Similar to the fully connected case, `b_grad` does not depend on `x` or `W`. We can compute `b_grad` only with the gradient at the output `y_grad`. Recall that each output map corresponds to a different element of the vector `b`. For backpropagation, we would need to sum the values of all of the gradient at the output `y_grad` values across the row, column, and batch dimensions. We do not need to sum over the output filter dimension n_o because there is a different value of `b` for each output filter. If the sum is done correctly, we should be left with a variable that is the same shape as `b`.

Deriving the gradients for $\mathbf{W}$ and $\mathbf{x}$ is more difficult than the fully connected case, because the convolution operation produces different output values for each output location. However, we can view a convolutional layer as a set of masked fully connected layers with weight sharing enforced. For example, consider one filter location. The output at one location can be expressed as $\mathbf{x}'\mathbf{W}'^T$, where $\mathbf{x}'$ is a row vector of the flattened data needed for one filter location, and $\mathbf{W}'$ is a row vector of the flattened kernel. We know from Lab 3 how to compute the backwards pass for this function. The only difference here is that the filter is sliding, so we need to sum over filter locations that are overlapping. This sum over locations operation can be expressed as a convolution.

Now we will consider $\frac{d\ell}{d\mathbf{x}}$. Every channel of the input affects every channel of the output. We can consider the contribution of each output channel to the input channels separately. We will use c_{out} to represent a particular output channel. The gradient due to a particular output channel $\frac{d\ell}{d\mathbf{x}_{c_{out}}}$ can be computed as a convolution between the output channel gradient $\frac{d\ell}{d\mathbf{x}^{l+1}_{c_{out}}}$, and the weight matrix for that channel $\mathbf{W}_{c_{out}}$. For this operation, we use a normal convolution instead of a correlation, because we are going in the "opposite" direction as the original operation. To obtain the actual gradient $\frac{d\ell}{d\mathbf{x}}$, we can sum the gradients due to each channel.

$$\frac{d\ell}{d\mathbf{x}} = \frac{d\ell}{d\mathbf{x}^{l+1}_{c_{out}}} ** \mathbf{W}_{c_{out}} \tag{2}$$

For computing $\frac{d\ell}{d\mathbf{W}}$ it is easiest to consider each input/output pair at a time. For a particular input/output pair, $\frac{d\ell}{d\mathbf{W}}$ is equal to the correlation between an input channel of $\mathbf{x}$ and an output channel of $\frac{d\ell}{d\mathbf{x}^{l+1}}$ (`y_grad`). For this convolution we assume that the input $\mathbf{x}$ is zero padded in the appropriate dimensions. A correlation with the padded $\mathbf{x}$ and $\frac{d\ell}{d\mathbf{x}^{l+1}}$ should give an output which is the same size as $\mathbf{W}$.

$$\frac{d\ell}{d\mathbf{W}}_{j,k} = \mathbf{x}_{channel=j} \overset{\text{corr}}{**} \frac{d\ell}{d\mathbf{x}^{l+1}}_{channel=k} \tag{3}$$

For these examples, we have left out an index i, corresponding to the sample index. As in Lab 3, all of the functions need to operate on a batch of data.

2.4 Initialization

As we talked in Lab 3, the initial values for the layer parameters must be set properly to achieve good network convergence. For this convolutional layer, we use same initialization strategy of Glorot [2] as we did for the fully-connected layer in Lab 3. The initialization in our network takes the form:

$$\mathbf{W} \sim \mathcal{N}\left(0, \sqrt{\frac{2}{f_{in} + f_{out}}}\right) \tag{4}$$

$$\mathbf{b} \sim \mathbf{0} \tag{5}$$

where

$$f_{in} = n_i \times h \times h \tag{6}$$

$$f_{out} = n_o \times h \times h. \tag{7}$$

Deliverable: conv.py

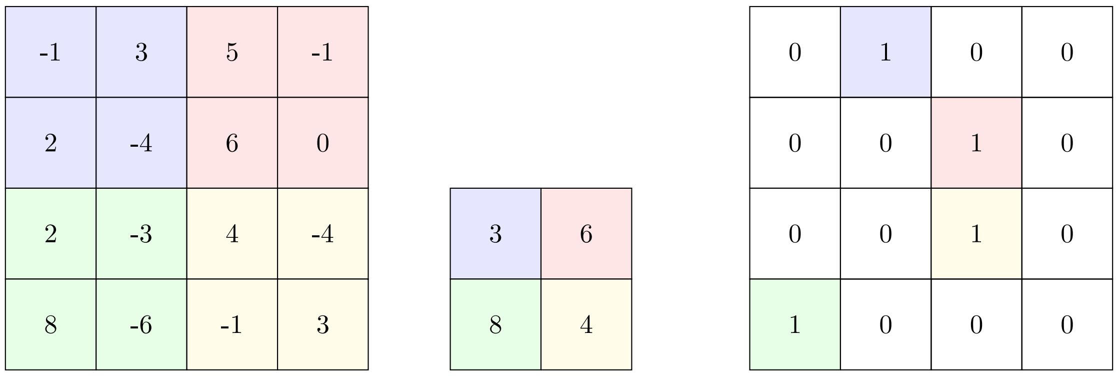
Submit a `conv.py` file that passes all of the tests in the `test_conv.py` file.

Extra Information: Convolution Speed

The speed of the convolution is often the bottleneck when training convolutional neural networks. Our implementation of the convolutional layer is likely to be several times slower than other library implementations, even when the other library is run on the CPU. Most libraries achieve this speedup by reformulating convolution as a matrix multiplication operation. As a matrix multiplication, efficient linear algebra libraries can be used for the convolution. Other methods for achieving faster convolutions include the Winnowgrad method [3], and FFT-based convolutions.

2.5 Update parameters

The `update_param` function is identical to that of the previous lab. The primary difference in this case is that our variables and gradients have more dimensions than the previous lab.



(a) Input **(b)** Output **(c)** Stored Locations

Figure 4: Maxpool forward pass for 2×2 max pooling.

3 Max Pooling Layer

A network that incorporates convolutional layers often uses Max Pooling layers as well. The max pooling operation simply takes the maximum activation over small regions in the input. For example, we could take the maximum value of each input over 2×2 sized spatial regions. There are two benefits to this operation. First, max pooling can add some robustness because it eliminates small activations and only takes the largest response in a spatial region. Another practical benefit of max pooling is that it can reduce the spatial dimensions of the inputs, which is similar to downsampling. The reduced spatial dimensions allow later convolutional layers to have a greater effective receptive field with respect to the input.

Forward pass We assume that the max pooling layer takes the maximum response using $k \times k$ sized square regions. There is no need to restrict the region to be square, however in your implementation you can assume that the region is square. Like in convolution, this region will move across the input to generate an output map. In general this region could move with some step size, which is often called the *stride*. For example unless a stride other than 1 is specified, in a typical convolution operation, the stride is one, meaning that the filter moves along every pixel. For the max pooling operation we won't use a stride of one. Instead we will use a stride that is of size k, the same size as our region. With this stride, the max pooling regions do not overlap. Figure 4 shows an example of this strided max pooling operation with $k = 2$.

The examples in Figure 4 assume a one channel input (and a one-channel output). In a typical network there will likely be an input with multiple channels. The max pooling layer should find the max in the region for each channel separately. The number of channels of the output should be the same as the number of channels of the input.

It is possible that the max pooling regions overlap with the edge of the input map. For example if the size of a dimension of the input map is odd, and the size of the max pooling region is 2×2, there will be a "left-over" position at the end. We will follow the convention of some machine libraries which ignore these left over positions.

Backward pass For the backward passes of the max pooling layers, we simply need to propagate the gradients to the positions that had the maximum activation value. In order to do this, we need to store the location of the maxima during the forward pass. Since the max pooling layer does not perform any arithmetic operations, the output gradient is passed exactly to the locations of the

6	2
-1	3

(a) Gradient from output

0	1	0	0
0	0	1	0
0	0	1	0
1	0	0	0

(b) Stored Locations

0	6	0	0
0	0	2	0
0	0	3	0
-1	0	0	0

(c) Gradient at input

Figure 5: Maxpool backwards pass for 2×2 max pooling.

maxima. For inputs that did not correspond to a maximum the gradient is 0. Figure 5 shows an example of this operation for $k = 2$.

For max pooling, it is possible that two (or more) inputs are both the max. In this case, max pooling should propagate the gradient for both of the inputs that were the max. This can occur often in neural networks, because the ReLU will force many inputs to be 0.

Deliverable: maxpool.py

Submit a `maxpool.py` file that passes all of the tests in the `test_maxpool.py` file.

4 Flatten Layer

Many network architectures are a combination of convolutional layers and fully connected layers. However the output of our convolutional layer is not compatible with the input of the fully connected layer from Lab 3. The convolutional layer output is of size (n_b, n_o, n_r, n_c) whereas the fully connected layer input should be of size (n_b, n_i). To correct for this we create a layer called `FlattenLayer` that "flattens" the last three dimensions of the convolutional layer such that the output of the `FlattenLayer` is of size $(n_b, n_o \times n_r \times n_c)$.

For the backwards pass, we simply need to apply the "inverse" flattening operation on the gradient. There are no learnable parameters or gradient calculations, it is simply another reshape operation. To perform the reshape for the forwards/backwards passes we can use the `np.reshape` function.

Hint: Tips in using `np.reshape` function

`np.reshape(x)` does not generate another instance of the variable x, but rather it generates a pointer to the variable x. Thus, if `y = np.reshape(x)`, changing the value of `y` will change the value of `x`. One solution is to use the `np.copy` function with the `reshape` function.

Deliverable: flatten.py

Submit a `flatten.py` file that passes all of the tests in the `test_flatten.py` file.

5 Putting it all together again

Finally, we will revisit training a neural network on a subset of the CIFAR-100 dataset. This time we will be using convolutional layers and maxpooling, in addition to ReLU and fully connected layers from the previous lab.

Important: Location for `cifar-100-python` folder:

You can download the CIFAR100 dataset using same link from Lab 3:
`https://www.cs.toronto.edu/~kriz/cifar-100-python.tar.gz`
But different from Lab 3, in Lab 4, after downloading the `cifar-100-python` folder, put it outside the `src` folder but in the same folder as the `src` folder.

We can reuse all of the layers from the previous lab. The only thing that might have to be modified is the ReLU layer, as the ReLU layer should be able to handle inputs from a fully connected layer, or a convolutional layer. Since the layers from the previous lab and this lab follow the same class interface, we can plug them into the same `Sequential` model that we developed in the previous lab. We could continue to develop different types of layers with the same interface and plug them into this model structure.

To classify CIFAR-100 images in the last lab, the data was flattened so that each data sample was a vector of pixel data. Here we do not want to flatten the data, because we want to use convolutional layers, which assumes that the input data has channels, rows, and columns. We have provided an updated `dataset.py` that provides the CIFAR-100 data in this form. The subset of 3 classes is determined using the seed parameter of the `cifar100` function. **Set this seed to your student ID number to get the subset of the data**.

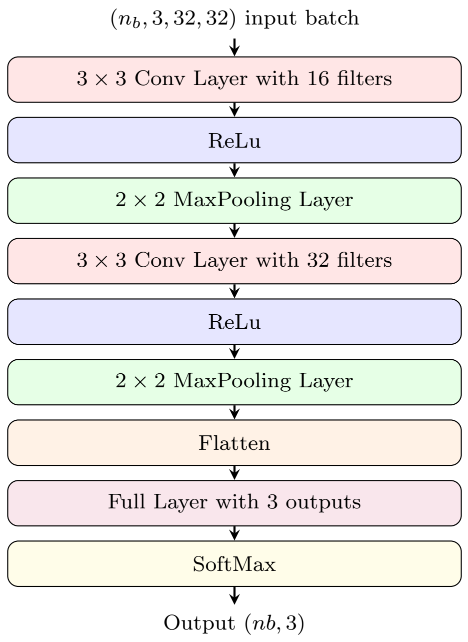
We will build a simple network with 2 convolutional layers and one fully connected layer. Figure 6 shows the architecture of this model. To keep the model from running too slow we will use small 3×3 sized filters. The small filter size is not a huge classification performance bottleneck, as high performing networks can be built entirely using 3×3 sized filters (e.g. [4]). Also to make training fast, we limit the number of filters in the convolutional layers to 16 and 32 units. After each convolutional layer, we apply a ReLU nonlinearity and a max pooling operation. The fully connected layer has an input size of 2048 which is equal to the size at the output of the second max pooling layer $(32 \times 8 \times 8)$. The final fully connected layer has 3 outputs, corresponding to the 3 classes.

Training may be slow due to the convolutional operation. So in this lab we do not require plots for different learning rates. However, we do require a plot of the training and validation loss for a learning rate of 0.1 for 15 epochs and a batch size of 128.

Deliverable: CIFAR100 ConvNet Results

Submit a script called `run_cifar100.py` that plots the average training loss vs epoch and that reports the final accuracy on the test set (please write the accuracy in the commented section at the beginning of the script).

Submit the plot of the average training loss vs epoch using a learning rate of 0.1 for 15 epochs and a batch size of 128.



$(n_b, 3, 32, 32)$ input batch
3×3 Conv Layer with 16 filters
ReLu
2×2 MaxPooling Layer
3×3 Conv Layer with 32 filters
ReLu
2×2 MaxPooling Layer
Flatten
Full Layer with 3 outputs
SoftMax
Output $(nb, 3)$

Figure 6: Network architecture.

6 Submission Instructions

We will test your code using unit tests similar to (but not identical to) the unit tests that we used in this lab. Thus you must be sure that your code works for all cases, not just the particular cases in the given unit tests. Do not change any of the names of the classes or functions. Also do not change the folder structure of the code as this is assumed by the testing code used for grading. You will be graded based on how many tests your code passes.

Replace the name of the given source folder with `FIRSTNAME_LASTNAME_LAB4` and zip the folder for submission. The plots that you generated during this lab should be placed in a folder called `results`.

The grading rubric for this lab is shown in Table 2.

Table 2: Grading rubric

Points	Description
10	Correct file names and folder structure (DO NOT submit the `cifar-100-python` folder)
30	`conv.py` implementation that passes all tests
30	`pool.py` implementation that passes all tests
15	`flatten.py` implementation that passes all tests
8	Plot of CIFAR100 subset training loss saved in `loss_plot.png`
7	`run_cifar100.py` that plots the training loss and report the accuracy
Total 100	

References

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

[2] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pp. 249–256, 2010.

[3] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4013–4021, 2016.

[4] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.