Lab 3 - A Neural Network from Scratch

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

1 Objectives

- Understand backpropagation
- Implement the fully connected Layer, ReLU layer, Softmax layer, and cross entropy loss function
- Create a network with the implemented layers, and test the implemented network on a subset of the CIFAR100 dataset

2 Understanding Backpropagation

Backpropagation is the method used to compute gradients in deep neural networks. The gradients allow us to use gradient descent based optimization algorithms to train deep networks. To understand backpropagation, we will first look at a simple example. Consider the functions:

$$f_1(x_1; w_1, b_1) = w_1 x_1 + b_1 \tag{1}$$
$$f_2(x_2; w_2, b_2) = w_2 x_2 + b_2 \tag{2}$$
$$z(x) = f_2(f_1(x)) \tag{3}$$

where x_i, w_i, and b_i are all scalars. Let's say we want to "train" this function to achieve a lower value of $\ell(x) = z(x)$. Obviously this function is trivial to minimize, but we can still see how to compute the gradients for all of the parameters using the backpropagation technique. Specifically, we need to compute $\frac{d\ell}{dw_1}$, $\frac{d\ell}{dw_2}$, $\frac{d\ell}{db_1}$, and $\frac{d\ell}{db_2}$. With these gradients we can use gradient descent to update the parameters.

We could ignore the decomposition of z into f_1 and f_2, write the full equation for z and derive the gradients manually. For this example, this method will work. However if we have a function composed of many other functions, as in a deep neural network, the final function derivatives will be very tedious to derive. It is usually much easier to compute the gradients of the smaller functions and build the gradient for the final function using the chain rule. This decomposition process will be the same that we use in backpropagation for training deep neural networks later in this lab.

Let's first consider the gradients in the f_1 function: $\frac{df_1}{dw_1}$, $\frac{df_1}{db_1}$, and $\frac{df_1}{dx_1}$. These are fairly easy to derive:

$$\begin{aligned} \frac{df_1}{dw_1} &= x_1 \\ \frac{df_1}{db_1} &= 1 \\ \frac{df_1}{dx_1} &= w_1 \end{aligned} \tag{4}$$

The gradients of the parameters in f_2 are identical to those in f_1, except for the different parameters w_2 and b_2. If we had many of these same functions composed together, we would only need to do these three gradient derivations, and we can reuse the derivations many times.

Now to find the gradients with respect to the final function z we can use the chain rule. Lets start with the parameters of f_2. These are the simplest gradients to compute because the function is closest to the output z.

$$\begin{aligned} \frac{d\ell}{dw_2} &= \frac{df_2}{dw_2} = x_2 = f_1(x_1) = w_1 x_1 + b \\ \frac{d\ell}{db_2} &= \frac{df_2}{db_2} = 1 \end{aligned} \tag{5}$$

Next we will compute the gradients of the parameters of f_1. Using the chain rule, we need to multiply the gradients of the f_1 parameters by $\frac{df_2}{dx_2}$. In other words, the gradients of the f_1 parameters depend on the computation of the gradient of the input to the f_2 function. The gradient flows backwards from the top of the function composition to the bottom. This gives the term *backpropagation*. Figure 1 shows this process. Using backpropagation we compute the gradients as follows:

$$\begin{aligned} \frac{d\ell}{dw_1} &= \left(\frac{df_2}{dx_2}\right)\left(\frac{df_1}{dw_1}\right) = w_2 x_1 \\ \frac{d\ell}{db_1} &= \left(\frac{df_2}{dx_2}\right)\left(\frac{df_1}{db_1}\right) = w_2 \end{aligned} \tag{6}$$

Now with all the gradients computed we can update the parameters using gradient descent to give a lower value of the loss ℓ. If this was a neural network, we could consider the functions f_1 and f_2 as layers of the network. Thus in backpropagation we compute the gradient of the last layer with respect to the input of the layer. The second to last layer uses this gradient to compute the gradient with respect to its own parameters. The gradient continues to flow down the network until we compute the gradient of the parameters of the first layer. For more details about the backpropagation algorithm, refer to Chapter 6 of [1].

Extra Information: Testing Gradients Numerically

When implementing the gradients of a function, or a complex nested function, it is useful to have a way to check whether our gradient derivation is correct. We can check our derivation numerically using the definition of the derivative:

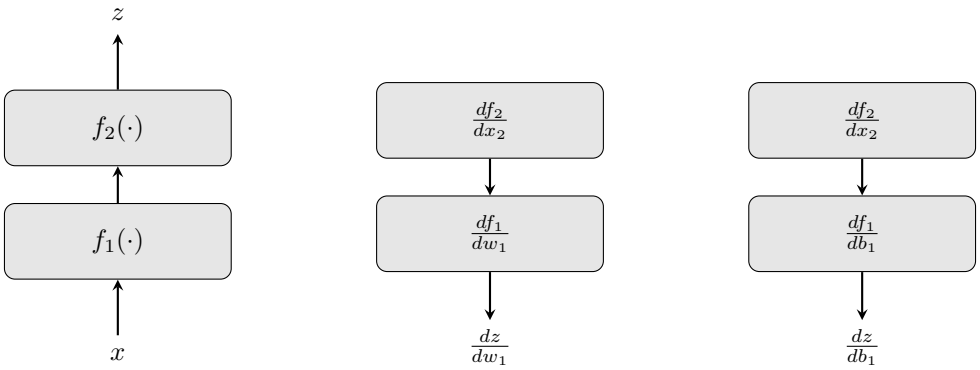
$$\frac{df}{da} = \lim_{h \to 0} \frac{f(a+h) - f(a)}{h}$$

This equation says that we change the parameter by a small value h. We then use the change in the output value to compute the derivative. If h is chosen to be small, the calculated gradient and the derived gradient should be very close. We use this type of check in the testing code provided in this lab to verify the implementation of deep network layers.

3 Implementing Layers of a Neural Network

With a basic understanding of backpropagation, we can build a simple neural network. We will be implementing the neural network as a collection of layer classes. We will implement classes for a fully connected layer, a ReLU layer, a softmax layer, and a cross entropy layer.

Each layer class implements an `__init__` function, a `forward` function, and a `backward` function. Layers with learnable parameters will additionally need a `update_params` function. The `__init__`



(a) Forward Pass **(b)** Backwards Pass for w_1 **(c)** Backwards Pass for b_1

Figure 1: Backpropagation example. In the forward pass (a) we pass the input x forward through the functions f_1 and f_2. In the backwards pass (b) we first need to compute $\frac{df_2}{dx_2}$, and then using chain rule we can obtain the desired $\frac{d\ell}{dw_1}$. If we included more functions (e.g. f_3), the process would be similar with more steps in the forward and backwards passes. Notice that in the backwards computation for the two parameters, we only need to compute $\frac{df_2}{dx_2}$ once. By utilizing this observation we can implement a fast version of backpropagation at the expense of memory utilization.

function initializes the layer parameters. The `forward` function implements the forward pass of the layer. The `backward` function of a layer l takes as input the gradient of the loss function with respect to the input of layer $l+1$ and computes the loss function's gradient with respect to the input and with respect of the learnable parameters of the considered layer l. Typically, the computed loss function's gradient with respect to the input of layer l is output to be used as input of the `backward` function of layer $l-1$, while the gradients with respect to the parameters are stored for later use in the gradient descent step. Finally, the `update_params` method will take some learning rate and use the computed gradients to update the values of the parameters. In the project folder we have provided templates for all of the layers that we need to implement (`full.py`, `relu.py`, `softmax.py`, `cross_entropy.py`).

Our implementation will be a Python package which we will name `layers`. As a Python package we can use it with `import` statements, just as we use the `numpy` library. A Python package can be easily reused in other projects. Creating a Python package called layers only requires the presence of a folder named `layers` with an `__init__.py` file, which we have provided. We put all of the implementation of our network in the `layers` folder. Then a file in the parent folder of the `layers` folder can import our network implementation with commands like:

```
import layers.FullLayer # can be used as: full = layers.FullLayer()
import layers.ReluLayer # can be used as: relu = layers.ReluLayer()
from layers import ReluLayer # can be used as: relu = ReluLayer()
from layers import FullLayer # can be used as: relu = FullLayer()
```

For now we will keep our package local, meaning that it can only be imported by files from the parent directory. It is possible to make the package global using `pip`. It is even possible to publish the package so that other users of `pip` can install the package from a network (but we won't be doing that here).

We have provided a set of tests for the neural network layers in the `test` folder. Different from previous labs, we have divided up the tests into individual files, where each file tests a single layer

class. In this way you can run the single test file for the particular layer that you are working on. You can also run all of the tests in the `test` folder by right clicking on the folder name and selecting `Run 'Unittests in test'` in PyCharm. You can add additional tests to ensure your code is correct, but do not modify the existing tests to make your code pass.

Extra Information: Speed

Loops in Python are slow, so if you want training to be fast, it is best to avoid them if possible. Numpy operations like `np.dot` call efficient LAPACK C libraries, so they are much faster than looping. The forward/backwards functions will be called thousands of times during the training, so if the forward/backward functions are very slow, the training will also be very slow.

Data Format and Batches We will be using stochastic gradient descent to train our network. In stochastic gradient descent, the dataset is split into batches of data points. The size of the batch is commonly 128 or 256, but there are works that use large batch sizes (e.g., > 1000) or very small batch sizes (e.g., 1). We run the forward and backwards passes of the network on the batch, and then update the parameters. This is different from Lab 2 where we computed the gradient on the entire training set.

The data passed between layers will be a matrix of size $n_b \times n_{f_i}$, where n_b is the batch size, and n_{f_i} is the number of features (outputs) in a particular layer i. n_{f_i} can change from layer to layer, but n_b will be the same throughout the entire network. This data organization means that a single data sample will be a row vector.

Our network will be able to do multi-class classification. To represent the class labels we will be using *one-hot encoding*. One-hot encoding represents the label as a vector of zeros with a one at the index of the label. For example if we have 5 classes, class 0 would be represented by the row vector $[1, 0, 0, 0, 0]$ and class 3 would be represented by $[0, 0, 0, 1, 0]$. Compared with representing the label as an integer, one-hot encoding simplifies the math and the implementation.

Layer Interface All of the layers will use the same class interface. In other words, they will all have the same function definitions with the same input parameter names. Only the implementation of each function varies for the different layers. With a common class interface, we can "plug-in" different layers to change a neural network architecture without needing any additional coding.

The initialization function (`__init__`) of the classes will need to set default values for the class variables. The initial values for network parameters can affect training convergence, so it is important to set these properly. Some network layers have hyper-parameters (such as the number of units in a fully connected layer). The settings of the hyper-parameters will be passed as arguments to the initialization function.

The `forward` function takes an input matrix `x` and returns some output. The input will always be of size $n_b \times n_{f_{i-1}}$ where n_b is the batch size, and $n_{f_{i-1}}$ is the dimensionality of the output from the previous layer.

The `backward` function takes in the input gradient from the higher layer (`y_grad`), returns the gradient with respect to the input, and stores the gradients of any trainable layer parameters. `y_grad` has shape $n_b \times n_{f_i}$ where $n_f f_i$ is the number of outputs of the layer. For some backwards functions we may need some computed values from the forward function. These computed values can be stored in class variables during the execution of the `forward` function.

The `update_param` method will take a learning rate `lr` and update the parameters of the layer. For layers without learnable parameters, `update_param` needs no implementation. We can use the Python command `pass` to implement a function that performs no operations. For other layers with

Function	Description
`__init__`	Initialization of class parameters
`forward`	Implement forwards pass of layer
`loss`	Implement backwards pass of layer
`update_param`	Update the learnable parameters of the layer

Table 1: Functions for each layer

learnable parameters, the stored gradients from the backwards function should be used to update the parameters.

3.1 Fully Connected Layer

Forward pass The fully connected layer is named as such, because each output of the fully connected layer is a weighted sum of each input element. The fully connected layer takes an input matrix of size $n_b \times n_i$ and outputs a matrix of size $n_b \times n_o$, where n_b is the batch size, n_i is the input size, and n_o is the output size. The weights for each output are learnable and stored in a matrix $\mathbf{W}$ of size $n_o \times n_i$. The implementation of the fully connected layer will be in `full.py`. The forward pass is defined by:

$$f_{full}(\mathbf{x}_i) = \mathbf{x}_i\mathbf{W}^T + \mathbf{b} \tag{7}$$

where $\mathbf{x}_i$ is a *row* vector for one sample of the input of the layer. In this lab we will be doing all the math with row vectors, so that the math can more easily transfer into code. In some textbooks or other resources you may see slightly different equations such as $\mathbf{W}\mathbf{x}_i$. The weight matrix is sometimes of size $n_i \times n_o$, and the vectors are sometimes expressed as column vectors. After the dimensions are taken into account, the different versions of the fully connected layer equation should express the same output. Eq. 7 describes the computation for one data sample. Our layer will need to handle a batch of data samples as the input, instead of only a single data sample.

Backwards pass Next, for the backwards pass we need to derive the Jacobians: $\frac{df_{full}}{d\mathbf{x}_i}$, $\frac{df_{full}}{d\mathbf{W}}$, and $\frac{df_{full}}{d\mathbf{b}}$. In this case the input and the output are both multi-dimensional vectors, so we actually need to compute Jacobian matrices. Recall that a Jacobian matrix is simply a matrix of the derivatives of every output with respect to every input.

$$\begin{aligned} \frac{df_{full}}{d\mathbf{x}_i} &= \mathbf{W} \\ \frac{df_{full}}{d\mathbf{b}} &= \mathbf{I}_{n_o} \end{aligned} \tag{8}$$

where $\mathbf{I}_{n_o}$ is the $n_o \times n_o$ identity matrix.

The Jacobian for $\frac{df_{full}}{d\mathbf{W}}$ is a tensor (multi-dimensional matrix) because we have an output of size n_o with respect to an input of size $n_o \times n_i$. The mathematics becomes a little difficult for this tensor Jacobian, however when we actually need to compute the gradient during backpropagation, the math simplifies greatly as we will see shortly. See [2] for a detailed derivation of this Jacobian.

These Jacobians give the gradient of the outputs of the layer relative to the inputs or parameters. However, what we actually need to compute is the gradient of the loss function (ℓ) with respect to these values. In the `backward` implementation, the input to the function `y_grad` gives the gradient of the output calculated by the later layer in the network. Mathematically `y_grad` represents $\left(\frac{d\ell}{d\mathbf{x}_i^{l+1}}\right)$, where $\mathbf{x}_i^{l+1}$ is the input to the next layer, or equivalently the output of the current layer. `y_grad` is of size $n_b \times n_o$. Similarly, we use $\mathbf{x}_i^l$ to denote the input to the current layer. We can take this

and use matrix multiplication with the Jacobians we just derived to get the gradient of the network output (or loss) relative to the parameter:

$$\begin{aligned}
\frac{d\ell}{d\mathbf{x}_i^l} &= \left(\frac{d\ell}{d\mathbf{x}_i^{l+1}}\right)\left(\frac{df_{full}}{d\mathbf{x}_i^l}\right) = \left(\frac{d\ell}{d\mathbf{x}_i^{l+1}}\right)\mathbf{W}^l \\
\frac{d\ell}{d\mathbf{W}^l} &= \sum_i \left(\frac{d\ell}{d\mathbf{x}_i^{l+1}}\right)\left(\frac{df_{full}}{d\mathbf{W}^l}\right) = \sum_i \left(\frac{d\ell}{d\mathbf{x}_i^{l+1}}\right)^T \mathbf{x}_i^l \\
\frac{d\ell}{d\mathbf{b}^l} &= \sum_i \left(\frac{d\ell}{d\mathbf{x}_i^{l+1}}\right)\left(\frac{df_{full}}{d\mathbf{b}^l}\right) = \sum_i \left(\frac{d\ell}{d\mathbf{x}_i^{l+1}}\right)
\end{aligned} \tag{9}$$

Note that here the gradients of the parameters sum over the data samples $\mathbf{x}_i$.

The `backward` function will return $\frac{d\ell}{d\mathbf{x}_i^l}$. This return value will be used by the `backward` function of the layer below the current layer. $\frac{d\ell}{d\mathbf{b}^l}$ and $\frac{d\ell}{d\mathbf{W}^l}$ should be stored in the `b_grad` and `w_grad` class variables. These stored values will later be used by the `update_param` function.

Updating the parameters The `update_param` function will update the parameters of the layer using the stored gradients computed in the `backward` function. We could have easily made this parameter update step a part of the `backward` function, but we leave it separate for flexibility. The PyTorch library, which we will see in a later lab, also separates the backwards pass from the parameter update step.

Recall from lab 2 that when we optimize parameters, we want to move in the opposite direction of the gradient such that the loss decreases. We use some learning rate (η) to control the speed of the descent. In general we could use different learning rates for different layers or parameters, but in our case we use the same learning rate for all of the learnable parameters.

$$\mathbf{b}' = \mathbf{b} - \eta\left(\frac{d\ell}{d\mathbf{b}}\right) \tag{10}$$

$$\mathbf{W}' = \mathbf{W} - \eta\left(\frac{d\ell}{d\mathbf{W}}\right) \tag{11}$$

Initialization The initial values for the layer parameters must be set properly to achieve good network convergence. If all the values of $\mathbf{W}$ are set equally, then the network won't learn interesting features because the values of $\mathbf{W}$ will all be updated with exactly the same values. To break this symmetry we can initialize $\mathbf{W}$ to have random values. The initialization of $\mathbf{b}$ is not as important. Setting $\mathbf{b}$ to a vector of zeros or small random numbers will work in most cases.

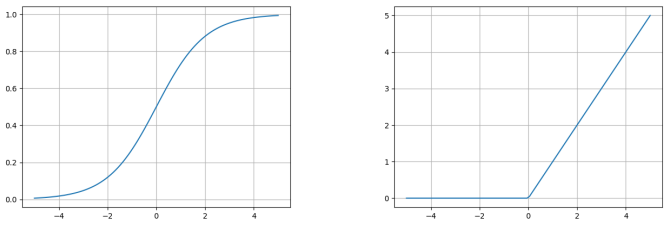
The scale of the random initialization of $\mathbf{W}$ can affect the training. For example, large initial values can lead to overflow. This is because each output of the fully connected layer is a weighted sum of all of the (potentially many) elements of the input sample. Glorot [3] found that a good initialization should depend on the number of inputs n_i and number of outputs n_o. n_i and n_o are sometimes called "fan-in" and "fan-out", respectively. Our network will use this type of initialization which takes the form:

$$\mathbf{W} \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_i + n_o}}\right) \tag{12}$$

$$\mathbf{b} \sim \mathbf{0} \tag{13}$$

3.2 ReLu Layer

To stack fully connected layers to create a deep network, we need to insert a non-linearity between the layers. If we do not insert the non-linearity, the two stacked fully connected layers are equivalent



(a) Sigmoid **(b)** ReLU

Figure 2: Non-linearity functions used for training neural networks. Historically, the sigmoid (a) was used, but most modern neural networks use the ReLU (b) function.

to a single fully connected layer. Thus without the non-linearity, the network does not have true depth, which may limit its ability to learn more complicated functions.

Historically, the sigmoid non-linearity was used. However, it was shown in [4] that the rectified linear unit (ReLU) nonlinearity works better for training neural networks. The ReLU is defined as:

$$f_{relu}(x_{i,j}) = \begin{cases} x_{i,j} & x_{i,j} > 0 \\ 0 & x_{i,j} \leq 0 \end{cases} \tag{14}$$

where $x_{i,j}$ is the element at index j of a single data sample $\mathbf{x}_i$. Figure 2 shows a plot of the ReLU nonlinearity.

For this layer we ask you to derive the equations needed for backpropagation yourself. The ReLU function mathematically does not have a derivative when $x_{i,j} = 0$. Similar to what we did in the SVM implementation from Lab 2, we can set the derivative at this point to 0.

3.3 SoftMax Layer

For classification problems, the last fully connected layer contains an output for each of the classes we are trying to learn. A larger output value means that the network has a stronger prediction for that particular class. However, the unbounded output values may not be very informative because there is no normalization of the output values. The softmax function will take these output values and normalize them such that the output distribution becomes a probability. The output probabilities will be more meaningful than the raw outputs of the last layer. The softmax function is defined as:

$$z_{i,j} = f_{softmax}(y_{i,j}) = \frac{e^{y_{i,j}}}{\sum_k e^{y_{i,k}}} \tag{15}$$

where $y_{i,j}$ is a single element of the input vector $\mathbf{y}_i$, which is the output of the last layer of the network. $\mathbf{y}_i$ is a row vector whose size is equal to the number of classes. In our implementation we need to compute the softmax for every element $y_{i,j}$ of every batch sample $\mathbf{y}_i$. We note that the softmax function looks very similar to the logistic function from Lab 2. In fact, the softmax function can be seen as a generalization of the logistic function to multiple classes.

Equation 15 is not numerically stable. One problem is that if y is large then $exp(y)$ will be very large. Dividing two large numbers can be numerically unstable because of limited precision in floating point arithmetic. In practice, we instead implement the softmax function by first subtracting the max of the data. Mathematically, the subtraction has no effect on the result, however it avoids the stability problems with large positive numbers.

$$y'_{i,j} = y_{i,j} - \max_{i,j}(y_{i,j})$$
$$z_{i,j} = f_{softmax}(y_{i,j}) = \frac{e^{y'_{i,j}}}{\sum_k e^{y'_{i,k}}} \tag{16}$$

For the backwards pass we need to compute the Jacobian matrix. For the softmax this is:

$$\frac{d\mathbf{z}_i}{d\mathbf{y}_i} = \text{diag}(\mathbf{z_i}) - \mathbf{z_i}^T\mathbf{z_i} \tag{17}$$

Again, like the other layers the softmax layer's `backward` function should take a gradient from the above loss layer and multiply with the Jacobian.

3.4 Cross Entropy

Since the softmax layer will transform the output values of the last fully connected layer into probabilities, we can use loss functions that assume probabilities as input. We will use *cross entropy* as our loss function. This cross-entropy function is similar to the cross entropy we used to train the logistic regression classifier in Lab 2. Cross entropy is usually preferred for classification tasks because of its relationship to KL-divergence and maximum likelihood. For our network, we use the multi-class version of cross entropy, because we want our network to be able to predict multiple classes. The multi-class cross entropy is defined as:

$$\ell(\mathbf{z}, \mathbf{t}) = -\frac{1}{n_b}\sum_i\sum_j \mathbf{t}_{i,j}\log(\mathbf{z}_{i,j}) \tag{18}$$

where $\mathbf{z}$ and $\mathbf{t}$ contain the softmax output and label for an entire mini-batch. $\mathbf{t}$ is assumed to be one-hot encoded. $\mathbf{y}_{i,j}$ is the jth output of the ith sample of the mini-batch, and $\mathbf{t}_{i,j}$ is the corresponding index in the one-hot encoding of the sample class. Log here refers to the base e logarithm.

The cross entropy layer does not have any learnable parameters, so we only need to compute the gradient with respect to the input of the layer. Additionally, we will assume that the cross entropy layer is the last layer in the network. There is no gradient coming from a higher layer, and so in the `backwards` function the input variable `y_grad` will be set to `None`. The `backwards` function for this layer simply needs to return the gradient given by:

$$\frac{d\ell(\mathbf{z}, \mathbf{t})}{d\mathbf{z}_{i,j}} = -\frac{1}{n_b}\frac{\mathbf{t}_{i,j}}{\mathbf{z}_{i,j}} \tag{19}$$

Deliverable: Network Layers

The following layers must be implemented with all unit tests passing:

- `relu.py`
- `full.py`
- `softmax.py`
- `cross_entropy.py`

4 Putting it all together

4.1 Implement the Sequential model

To put all of the layers together we will implement another class that collects all of the layers to form a deep neural network. Similar to the Scikit-Learn classes, and the classes we implemented in Lab 2, the deep network class will implement a `fit(x,y)` function and a `predict(x)` function. We call our model `Sequential` as our model implements a sequential list of layers. Our simple model won't support any connections that are not sequential (i.e., we do not support residual connections as in [5]). We could implement a different model that connects the layers in a more flexible directed graph, but we leave this exercise for the interested reader.

The `Sequential` model will be initialized with a list of layers and a loss layer. We will design the model such that it is generic and will accept an arbitrarily long list of layers. For example we can define our sequential model as follows:

```
from layers import (FullLayer, ReluLayer, SoftMaxLayer,
                    CrossEntropyLayer, Sequential)

model = Sequential(layers=(FullLayer(32*32*3, 500),
                           ReluLayer(),
                           FullLayer(500, 5),
                           SoftMaxLayer()),
                   loss=CrossEntropyLayer())
```

This code defines a network with 2 learnable fully connected layers. In between the fully connected layers is a Relu layer, and at the end of the second fully connected layer is a soft-max layer. The first fully connected layer has an input size of $32 \times 32 \times 3$ which could correspond to a flattened 32×32 pixel color image and an output size of 500. The output size here is a designed hyper-parameter of the network. The next fully connected layer has an input size of 500. For our network implementation, the input size of a fully connected layer must be the same as the output size of the previous layer. The output size of the second fully connected layer is 5, which corresponds to 5 output classes. Finally, we must also define the loss function when we initialize the model.

The sequential model is much like a layer, in that it contains `forward` and `backward` functions of its own. These functions will simply call all of the layers' `forward` or `backward` functions and make sure the data is appropriately passed between the layers. We have provided the template in `sequential.py` and tests in `test/test_sequential.py`.

The `forward` function Like a layer, the sequential model has a `forward` function. This function will take the input and pass it through the first layer. The output of the first layer should be passed to the second layer, and so on until we obtain the output of the last layer. This function should work with an arbitrary number of layers that are stored in the `self.layers` class variable. We allow the `forward` function to either return the output of the last layer, or the output of the loss function, depending on whether the optional `target` variable is passed. This allows us to use the `forward` function both for computing the loss and for obtaining the output prediction during testing.

The `backward` function The sequential model should also have a `backward` function that will go through all of the layers for the backwards computation of the gradients. For this function, we start with the loss layer and compute the gradient. The `backward` function of the loss layer can be called without any inputs. Then we feed the loss layer gradient as input to the `backward` function of the previous layer. The output of that latter backward function is again fed as input to the backwards function of the preceding layer, and so forth. The value of a layer gradient is

computed using the value of the input of that layer, which is obtained from the values stored by the `forward` function. We keep doing this until we call the `backward` function of the first layer. In our implementation of the `forward` function for the layers, we should have saved all of the values needed in the `backward` functions. After the backwards pass, the gradients of each parameter will be stored by the corresponding layers to be updated.

The `update_params` function We also need a simple function that will update the parameters of the network by some learning rate. For this function we don't need to worry about passing data between layers. We can simply call the `update_params` function of each layer in the network.

The `fit` function Finally we can implement the `fit` method to enable our model to be trained. This method will be similar to the `fit` function that we implemented for the logistic regression classifier in Lab 2 that used gradient descent.

In Lab 2, we computed the gradient on the entire input data, and then updated the parameters. For the `Sequential` class, we will assume that the size of our data set is very large (> 1000 samples). In this case it may be impossible to calculate the gradients at once as we did in Lab 2, because of memory limitations. Instead we will split the data into smaller batches of images. These batches are often also called mini-batches. This method of batch-wise training is called *stochastic gradient descent*. Besides saving memory, it also has some theoretical advantages over training using the entire dataset.

For a particular mini-batch we need to perform several operations for training. First, we use the `forward` function to do the forward pass of the network for the particular batch. After this pass, the layers should store all the necessary outputs used in the backwards computation. Next we call the `backward` function to compute the gradients of all of the layers. Finally, we can call the `update_params` function to update the parameters of the network, based on this particular batch.

It is important that the mini-batch is a decent sampling of the actual dataset. If the mini-batch consisted of only a single class, then the network won't learn differences between classes. The easiest way to prevent this is to shuffle the data before splitting into mini-batches. In our case, we assume that the data is already shuffled before calling the fit function.

The `predict` function Most of the work in the `predict` function is done by the previously mentioned `forward` function. The `forward` function gives the estimated probability of each class for each input data sample. We want our `Sequential` class to be a classifier, so it should return class labels instead of probabilities for each input data sample. In Lab 2, we obtained labels from the probabilities by using a threshold of 0.5. In this case, there are several output classes, so we want to return the class that has the highest predicted probability. We assume that the `predict` function is called on a small batch of data. So we do not need to worry about splitting the data into batches for this function.

4.2 Training the model with CIFAR100 data

Now, assuming all of the tests are passing, we can attempt to train our model on a subset of the CIFAR100 dataset [6]. This dataset consists of 60,000 32×32 pixel color images for 100 classes. Although the images are of a very small size, humans can still recognize the class of the image (Figure 3). The CIFAR datasets are often used for machine learning research because the small image size makes training fast, but there is still a large number of images so we can fit large models.

For this lab, we will train and test on a 4-class subset of the CIFAR100 dataset. The restriction to 4 classes makes the network faster to train, as well as makes the problem a little easier for our network to learn. Each individual will train/test on a different subset of the dataset. Testing accuracies of different sets can vary since different class combinations can be more difficult.

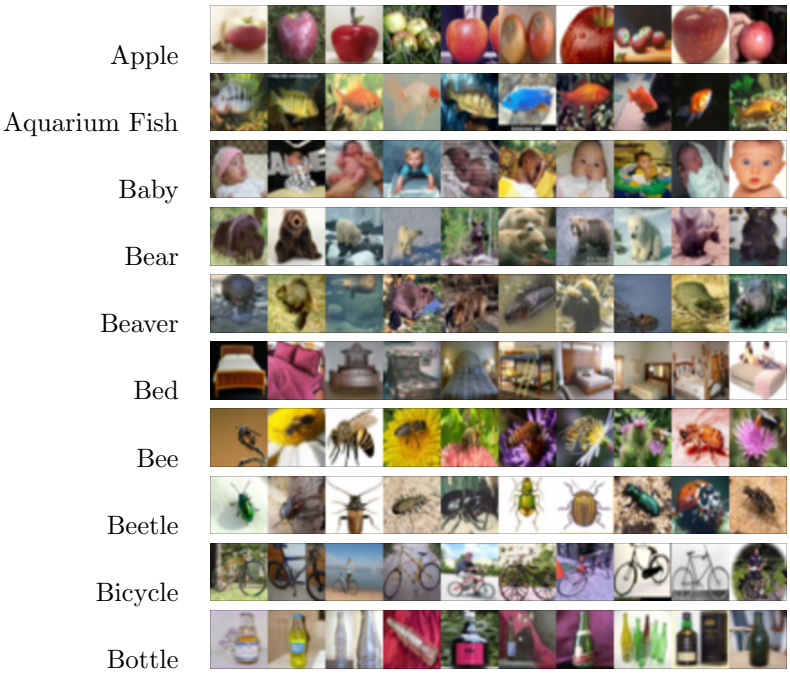


Figure 3: Example classes from CIFAR100 dataset. For viewing, images are enlarged from the original 32×32 size.

We have provided a function that will load the CIFAR100 dataset in the `dataset.py` file. This file loads the CIFAR100 data from the a `cifar-100-python` folder. The `cifar-100-python` folder can be downloaded at the following link:

https://www.cs.toronto.edu/~kriz/cifar-100-python.tar.gz

The subset of the CIFAR100 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 scale the image data to be between -0.5 and 0.5. It is generally a good idea to scale your data before training a network, because some parameter initializations such as Glorot assume that the inputs have zero mean. You can load the dataset with the following code:

```
from layers.dataset import cifar100
(x_train, y_train), (x_test, y_test) = cifar100()
```

You will need to take this data and filter it such that it only has the data corresponding to the assigned subset.

Now, assuming all of the tests are passing, we can try to train our model. For this lab we will be training a simple 2-layer neural network. The input to the network is the raw image pixels. We will not be considering any spatial relationships between pixels (as in convolutional neural networks), so we can simply flatten the $32 \times 32 \times 3$ image into a 3072 length vector for the input of our network. The first layer has 500 units, so the output of the first layer will be of length 500. After the first layer, we use a ReLU layer to add nonlinearity to the network. After the ReLU layer, we use another fully connected layer that outputs 5 values, corresponding to the 5 classes of our classification problem. These outputs will be fed into a softmax layer to output final class probabilities. We will use the cross entropy function as our loss function. This network structure can be built with the code presented in Section 4.1.

For each mini-batch we get the loss from the `forward` function. If we plot this loss for each mini-batch, we will see a noisy, but generally decreasing plot. The plot is noisy because the different (random) mini-batches have different samples that may be easier or more difficult than the average sample in the dataset. A less noisy estimate of model loss can be found by averaging all of the mini-batch losses over an entire epoch. This average loss is different than the loss over the entire dataset, because we update the weights after every mini-batch, so the true epoch loss changes after each mini-batch update. The average mini-batch loss is usually a good approximation of the loss of an epoch, and is generally preferred over recomputing the epoch loss because of the computational savings. When we ask you to plot the loss for an epoch, you can use this average of mini-batch losses.

Deliverable: Sequential model with CIFAR100 results

- Implementation in `sequential.py` that passes all tests
- Plot the average training loss vs epoch for learning rates 0.1, 0.01, 10 for 15 epochs
- Use the sequential model to construct a neural network that achieves higher testing accuracy than the given network architecture. You can try adding more layers and changing the width (number of outputs) of each layer. Submit a file that trains and tests this model as `better_cifar.py`.

Hint: Debugging

If the unit tests pass, this is some indication that your code is correct. However, the unit tests do not verify everything that could potentially go wrong during training. When training deep networks, the code can be correct, but the training may still not converge. This could be because of poor parameter initialization, exploding/vanishing gradients, lack of data normalization, insufficient data, etc.

5 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_LAB3` 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.

References

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[2] J. Johnson, "Backpropagation for a Linear Layer." http://cs231n.stanford.edu/handouts/linear-backprop.pdf. [Online; accessed 22-Dec-2017].

Table 2: Grading rubric

Points	Description
10	Correct file names and folder structure
15	`full.py` implementation that passes all tests
10	`relu.py` implementation that passes all tests
15	`softmax.py` implementation that passes all tests
15	`cross_entropy.py` implementation that passes all tests
15	`sequential.py` implementation that passes all tests
10	Plot of CIFAR100 subset training loss and testing classification accuracy as a function of learning rate
10	`better_cifar.py` that trains and tests a network that achieves better accuracy than the given network
Total 100	

[3] 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.

[4] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 807–814, 2010.

[5] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[6] A. Krizhevsky, "Learning multiple layers of features from tiny images," 2009.