

Due: Wednesday, April 23 at 11:59 pm (Start EARLY!)

Deliverables:

1. Submit your predictions for the test sets to Kaggle as early as possible. Include your Kaggle scores in your write-up (see below). The Kaggle competition for this assignment can be found at

- <https://www.kaggle.com/t/8eabb9826c5a4679bd6cb966144a2638>

2. The written portion:


- Submit a PDF of your homework, **with an appendix listing all your code**, to the Gradescope assignment titled “Homework 6 Write-Up”. Please see Section 3.5 for an easy way to gather all your code for the submission (you are **not** required to use it, but we strongly recommend it).
- In addition, please include, as your solutions to each coding problem, the specific subset of code relevant to that part of the problem. Whenever we say “include code”, that means you can either include a screenshot of your code, or typeset your code in your submission (using markdown or \LaTeX).
- You may typeset your homework in LaTeX or Word (submit PDF format, **not** .doc/.docx format) or submit neatly handwritten and scanned solutions. **Please start each question on a new page.**
- If there are graphs, include those graphs in the correct sections. **Do not** put them in an appendix. We need each solution to be self-contained on pages of its own.
- In your write-up, please state with whom you worked on the homework.
- In your write-up, please copy the following statement and sign your signature next to it. (Mac Preview and FoxIt PDF Reader, among others, have tools to let you sign a PDF file.) We want to make it *extra* clear so that no one inadvertently cheats.
“I certify that all solutions are entirely in my own words and that I have not looked at another student’s solutions. I have given credit to all external sources I consulted.”

3. The coding portion:

- Submit all the code needed to reproduce your results to the Gradescope assignment entitled “Homework 6 Code”. Yes, you must submit your code twice: in your PDF write-up following the directions as described above so the readers can easily read it, and once in the format described below so the assignment autograder can parse them correctly.
- Your submission should include all the files in the [neural networks package](#). Simply drag and drop these files directly into the Gradescope submission box; don’t package them in a directory or zip them in any way.
- Do **NOT** submit any data files that we provided.
- Please also include a short file named README listing your name, student ID, and instructions on how to reproduce your results.

- Please take care that your code doesn't take up inordinate amounts of time or memory. If your code cannot be executed by the autograder, your solution cannot be verified.
- Finally, also include any code you write for the [PyTorch](#) section of this homework into your Gradescope submission. If you choose to work directly in Google Colab, feel free to simply download your python notebook and attach it with the rest of your submission.
- At the end, your Gradescope submission screen might look like the following.

Submit Programming Assignment

 Upload all files for your submission

Submission Method

☒  Upload ☐  GitHub ☐  Bitbucket

Add files via Drag & Drop or [Browse Files](#).

Name	Size	Progress	x
__init__.py	0 b	<div></div>	x
activations.py	5.8 KB	<div></div>	x
datasets.py	4.2 KB	<div></div>	x
layers.py	17.3 KB	<div></div>	x
logs.py	3.1 KB	<div></div>	x
losses.py	1.9 KB	<div></div>	x
models.py	9.6 KB	<div></div>	x
optimizers.py	2 KB	<div></div>	x
schedulers.py	1.3 KB	<div></div>	x
utils.py	2.9 KB	<div></div>	x
weights.py	5.4 KB	<div></div>	x
CS189_HW6_NN.ipynb	28.5 KB	<div></div>	x
README.md	0 b	<div></div>	x
train_ffnn.py	3.2 KB	<div></div>	x

1 Honor Code

Declare and sign the following statement:

“I certify that all solutions in this document are entirely my own and that I have not looked at anyone else’s solution. I have given credit to all external sources I consulted.”

Signature : _____

While discussions are encouraged, *everything* in your solution must be your (and only your) creation. Furthermore, all external material (i.e., *anything* outside lectures and assigned readings, including figures and pictures) should be cited properly. We wish to remind you that consequences of academic misconduct are *particularly severe*!

2 Background

This section will provide a background on neural networks that is designed to help you complete the assignment. There are no questions in this part. The questions for the homework begin in Section 4.

2.1 Neural Networks

Many of the most exciting recent breakthroughs in machine learning have come from “deep” (many-layered) neural networks, such as the [deep reinforcement learning algorithm](#) that learned to play Atari from pixels, or the [ChatGPT](#) model, which generates text that is nearly indistinguishable from human-generated text.

Neural network libraries such as TensorFlow, PyTorch, and JAX have made training complicated neural network architectures very easy. However, we want to emphasize that neural networks begin with fundamentally simple models that are just a few steps removed from basic logistic regression. In this assignment, you will build two fundamental types of neural network models, all in plain `numpy`: a **feed-forward fully-connected network**, and a **convolutional neural network**. We will start with the essential elements and then build up in complexity.

A neural network model is defined by the following.

- An **architecture** defining the flow of information between computational layers. This defines the composition of functions that the network performs from input to output.
- A **cost function** (e.g. cross-entropy or mean squared error).
- An **optimization algorithm** (e.g. stochastic gradient descent with backpropagation).
- A set of **hyperparameters**. (Here we use this as a catch-all term to also include algorithm parameters that technically are not “hyperparameters” in the traditional sense because they don’t change the bias or the variance, such as the learning rate and the minibatch size for stochastic gradient descent with minibatches.)

Each *layer* is defined by the following components.

- A **parameterized function** that defines the layer’s map from input to output (e.g. $f(x) = \sigma(Wx + b)$).
- An **activation function** σ (e.g. ReLU, sigmoid, etc.).
- A set of **parameters** (e.g. weights and bias terms).

Neural networks are commonly used for supervised learning problems, where we have a set of inputs and a set of labels, and we want to learn the function that maps inputs to labels. To learn this function, we need to update the parameters of the network (i.e. the weights, including the bias terms). We do this using **minibatch gradient descent**. To compute the gradients for gradient descent, we use a dynamic programming algorithm called **backpropagation**.

In the backpropagation algorithm, we first compute what is called a **forward pass** of the network. In the forward pass, we send a minibatch of input data (e.g. 50 training points) through the network. The output is a set of predicted labels, which we use as input to our loss function (along with the true labels from the training data). We then take the gradients of the loss with respect to the parameters of each layer, starting with the output of the network and using the chain rule to propagate backwards through the layers. This is called the **backward pass**. During the backward pass we compute the gradients of the loss function with respect to each of the model parameters, starting from the last layer and “propagating” the information from the loss function backwards through the network. This lets us calculate gradients with respect to all the parameters of our network while letting us avoid computing the same gradients multiple times.

To summarize, training a neural network involves three steps.

1. Forward propagation of inputs.
2. Computing the cost.
3. Backpropagation and parameter updates.

2.2 Minibatches

When building neural networks, we have to carefully consider the data. In Homework 4, you coded both batch gradient descent and stochastic gradient descent for logistic regression. For the stochastic version, where only a single data point was used, the form of gradients used in gradient descent were different than those of batch gradient descent. Neural networks usually operate on minibatches, or subsets of the data matrix. This is because iterating on all the data at once (batch gradient descent) is inefficient for large data sets, whereas iterating on just one training point at a time introduces excessive stochasticity (randomness) and makes poor use of your computer’s caches and potential for parallelism. Thus, every step of your neural network should be defined to operate on minibatches of data. During a single operation of minibatch gradient descent, you take a matrix of shape (B, d) where B is the minibatch size and d is the number of features, and do a forward pass on B training points at once—ideally using vector operations to obtain some parallelism in your computations (as every training point is processed the same way). The input to a convolutional neural network for image recognition might be a four-dimensional array of shape (B, H, W, C) where B is the minibatch size, H is the height of the image, W is the width of the image, and C is the number of channels in the image (3 for RGB—that is, red, green, and blue intensities).

As you are writing the gradient descent algorithm to work on minibatches, all of your derivations must work for minibatches. For that reason, many of the derivations you do for this homework will differ from those you have seen in class. Thinking in terms of minibatches often changes the shapes and operations you do. **Your derivations must be for minibatches and cannot use loops to iterate over individual data points.** Be prepared to spend some time working out the tricky details of how to do this.

2.3 Feed-Forward, Fully-Connected Neural Networks

A feed-forward, fully-connected neural network consists of layers of units alternating with layers of edges. Each layer of edges performs an **affine transformation** of an input, followed by a nonlinear activation func-

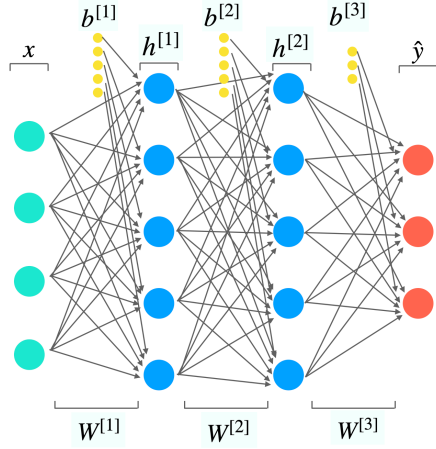


Figure 1: A 3-layer fully-connected neural network.

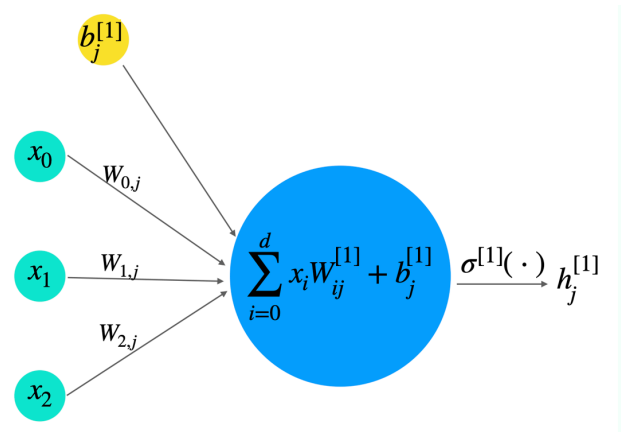


Figure 2: A single fully-connected neuron.

tion. “Fully-connected” means that a layer of edges connects every unit in one layer of units to every unit in the next layer of units. We use the following notation when defining fully-connected layers, with superscripts in brackets indexing layers (both layers of units and layers of edges) and subscripts indexing the vector/matrix elements. In this notation, we will use **row vectors** (not column vectors) to represent unit layers so that we can apply successive matrices (edge layers) to them from left to right.

- x : A single data vector, of shape $1 \times d$, where d is the number of features. You can think of it as “unit layer zero.” We present a training point or a test point here.
- \hat{y} : A single output vector, of shape $1 \times k$, where k is the number of output units. These could be regression values or they could symbolize classifications (and you can mix output units of both types). Each training point x is accompanied by a label vector y , and the goal of training is to make x ’s output \hat{y} be close to y .
- $n^{[l]}$: The number of units (neurons) in unit layer l .
- $W^{[l]}$: A matrix of weights connecting unit layer $l-1$ with unit layer l , of shape $n^{[l-1]} \times n^{[l]}$. This matrix represents the weights of the connections in edge layer l . At edge layer 1, the shape is $d \times n^{[1]}$.
- $b^{[l]}$: The bias vector for layer l , of shape $1 \times n^{[l]}$.
- $h^{[l]}$: The output of edge layer l . This is a vector of shape $1 \times n^{[l]}$.
- $\psi^{[l]}(\cdot)$: The nonlinear *activation function* applied at layer l .

A fully-connected layer l is a function

$$\phi^{[l]}(h^{[l-1]}) = \psi^{[l]}(h^{[l-1]}W^{[l]} + b^{[l]}) = h^{[l]}.$$

The output $h^{[l]}$ of edge layer l is computed then used as the input to edge layer $l+1$. (At edge layer 1, $h^{[0]}$ is simply the input vector x .) A neural network is thus a composition of functions. We want to find weights such that the network maps each training point x to its label y .

In a multiclass classification problem with more than two classes, it is common to set k equal to the number of classes and have each output unit represent a true/false value for one class. This is called *one-hot encoding*. A one-hot encoded output unit used for classification might employ the labels

$$y_i = \begin{cases} 1 & x \in \text{class } i, \\ 0 & \text{otherwise.} \end{cases}$$

For example, for a classification problem with 3 classes, the label for a training point in class 3 might be $(0, 0, 1)$ and the label for a training point in class 2 might be $(0, 1, 0)$. However, the precise values you choose ought to depend on the activation function ψ . Moreover, for reasons explained in lecture, you might get better results by using less extreme labels, such as 0.05 and 0.95, in lieu of 0 and 1, if s is the logistic (sigmoid) function. If you have only two classes, there is usually no advantage to one-hot encoding; one output unit for the class label should suffice.

2.4 Batch Normalization

Neural network training can often be sped up by normalizing the training data. The simplest normalization is to center and scale each input feature, so each feature has mean zero and variance one. This tends to improve the conditioning of the cost function we optimize. Could we do the same thing with the hidden unit values? Unlike the input data, we don't know in advance what the mean or variance of the hidden unit values will be.

This was the inspiration for *batch normalization*, invented by Ioffe and Szegedy in 2015. Batch normalization is often very effective at both speeding up training and improving generalization (that is, obtaining lower test error), especially for very deep networks. Unfortunately, there is still controversy and confusion over why batch normalization works, and it appears to be broadly accepted that the explanation given by Ioffe and Szegedy in their original paper has been firmly refuted. Nevertheless, batch normalization works so well that it is used very widely in practice, and nobody cares that we aren't sure why. Many of the staple networks of deep learning, such as [ResNet](#) and [VGG](#), use batch normalization¹.

Consider a layer of hidden units represented by a vector h . Batch normalization adds a layer to the network that performs a linear transformation that “normalizes” each component of h , yielding another vector z , which replaces h as the input to the next layer of the network.

Batch normalization updates that linear transformation with every minibatch. Let X be a $B \times d$ design matrix representing one minibatch of B training points (chosen randomly from the training set), and let X_i be the i th training point in the design matrix (i.e., the i th row of X is X_i^T). When we present a minibatch point X_i at the input of the neural network, we compute a corresponding vector h of hidden unit values. We compute the statistics over the minibatch of h 's components. That is, let $\tilde{\mu}$ be the average value of h over the minibatch—so $\tilde{\mu}_i$ is the average value of hidden unit h_i —and let $\tilde{\sigma}_i^2$ be the variance of h_i over the minibatch.

$$\tilde{\mu} = \frac{1}{B} \sum_{j=1}^B h^{(j)}, \quad \tilde{\sigma}_i^2 = \frac{1}{B} \sum_{j=1}^B (h_i^{(j)} - \tilde{\mu}_i)^2.$$

Now we can normalize the hidden unit values for each training point in the minibatch by setting

$$\tilde{h}_i = \frac{(h_i - \tilde{\mu}_i)}{\sqrt{\tilde{\sigma}_i^2 + \epsilon}},$$

¹The original VGG network doesn't use batch normalization, but the newer versions do, and they have better performance.

where ϵ is a small value included to dodge the case where $\hat{\sigma}_i^2$ is zero or very close to zero. Observe that each hidden unit is normalized independently of all the others.

There's one more twist. Who says that the ideal mean is zero, and the ideal variance is one? Maybe some other mean and variance would work better. The radical idea of batch normalization is to let the neural network learn the best value of the mean and variance for each hidden unit. The output of a batch normalization layer is a vector z with the components

$$z_i = \gamma_i \tilde{h}_i + \beta_i,$$

so we are trying to get the component z_i to have a mean (over all possible inputs) of roughly β_i and a variance of roughly γ_i^2 . Note that z , β , and γ are vectors of the same length as h . (Because Ioffe and Szegedy used γ and β in their paper.)

The statistics of the minibatch might not be a good estimate of the statistics of the entire training set. (Moreover, the training set might be infinite, as it typically is if we use data augmentation techniques.) After training is finished, at test time, we want to use estimates for the entire training set. Therefore, we maintain a long-term estimate of the component means and variances. Specifically, we maintain an exponential moving average: prior to the first minibatch we set $\hat{\mu} \leftarrow 0$ and $\hat{\sigma}_i^2 \leftarrow 0$, and for each subsequent minibatch we apply the update rule ²

$$\hat{\mu} \leftarrow \alpha \tilde{\mu} + (1 - \alpha) \hat{\mu}, \quad \hat{\sigma}_i^2 \leftarrow \alpha \tilde{\sigma}_i^2 + (1 - \alpha) \hat{\sigma}_i^2,$$

for some suitable $\alpha \in (0, 1)$. To conform with PyTorch's implementation, we set the momentum term as $1 - \alpha$ in the operation, which you will see in the code.

We collect these notations in one place. The parameters to be learned are the vectors β and γ .

- X : $B \times d$ design matrix representing one minibatch. Each row is a training point.
- h : Vector of hidden unit values for one training point X_i in the minibatch; input to the batch normalization layer; a vector with shape $1 \times d$.
- z : Output of the batch normalization layer; a vector with shape $1 \times d$.
- $\tilde{\mu}$: The sample mean of h for the minibatch.
- $\tilde{\sigma}_i^2$: The sample variance of h_i for the minibatch.
- $\hat{\mu}$: The moving average of h for all training points so far.
- $\hat{\sigma}_i^2$: The moving variance of h_i .
- β : The target mean for h . This is a learnable vector with shape $1 \times d$, which means that you'll need to compute the gradient $\nabla_{\beta} L$.
- γ_i : The target variance for h_i . γ is also a learnable vector of shape $1 \times d$.

When training is finished, we freeze the values of β and γ_i , and use the running mean and variance $\hat{\mu}$ and $\hat{\sigma}_i$ for use when test points are presented to the network.

²We do not set the running mean and variance to the mean and variance of the first minibatch to conform with PyTorch's implementation. Also, once you have passed in enough minibatches, your choice to initialize will give very minimal differences.

2.5 Convolutional Neural Networks

We will use the following notation when defining a convolutional neural network layer. The weights that the neural network needs to learn are $W^{[l]}$ and the bias terms $b^{[l]}$.

- X : The network's input, a single image tensor (multi-dimensional array), of shape $d_1 \times d_2 \times c$, where d_1 and d_2 are the spatial dimensions, and c is the number of channels (of which there are typically three, encoding red, green, and blue pixel intensities).
- \hat{y} : The network's output, a single vector of shape $1 \times k$.
- $n^{[l]}$: The number of channels in layer l .
- $(k_1, k_2)^{[l]}$: The size of the spatial dimensions of each filter/mask/kernel in layer l . Sometimes called the *kernel size*.
- $W^{[l]}$: A four-dimensional array (sometimes called a “tensor”) collecting all the filters at edge layer l . This tensor has shape $k_1 \times k_2 \times n^{[l-1]} \times n^{[l]}$.
- $b^{[l]}$: The bias vector for layer l , of shape $1 \times n^{[l]}$.
- $H^{[l]}$: The output of layer l . This is a tensor of shape $r_1 \times r_2 \times n^{[l]}$, where (r_1, r_2) is the shape of output of the convolution operation. Below we will discuss how to calculate this.
- $\psi^{[l]}(\cdot)$: The nonlinear *activation function* applied at layer l .

In a convolutional layer, each filter is convolved with the input image, across every image channel. This operation is, essentially, a sliding sum of elementwise products. Figure 3 gives a visual example. Let Z denote the intermediate result *just before we apply the activation function* ψ to obtain H . To compute a single element in the intermediate output Z , for a single unit n , we compute

$$Z[d_1, d_2, n] = (X * W)[d_1, d_2, n] = \sum_i \sum_j \sum_c W[i, j, c, n] X[d_1 + i, d_2 + j, c] + b[n].$$

Note: this formula is the **cross-correlation** formula from signal processing and NOT the convolution formula. Nevertheless this is what ML people call convolution, and so will we. It actually makes sense to say cross-correlation instead of convolution because the former can be interpreted as producing an output which is higher at locations where the image has the pattern in the filter and low elsewhere. Convolution is the same as cross-correlation with a flipped filter, and our filters are learned, so it makes no difference operationally whether you implement convolution or cross-correlation. However, to pass our tests, you must implement **cross-correlation** and call that convolution because that's how we do it in ML-land.

In this equation, we drop the layer superscripts for clarity, and index elements of the matrices in brackets. The pre-activation Z is what we call a *feature map*, which essentially captures the strength of each filter at every region in the image. In the equation above, we slide the filter over the image in increments of one pixel. We can choose to take larger steps instead. The size of the step taken in the convolution operation is referred to as the *stride*.

The output of the convolutional layer is

$$H^{[l]} = \psi^{[l]}(Z^{[l]}).$$

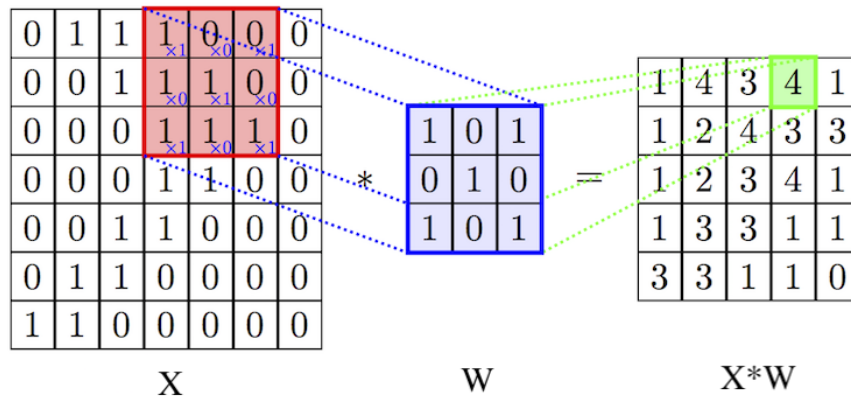


Figure 3: Figure showing an example of one convolution. Note that if we have more than 1 channel (such as an RGB image), we accommodate this by matching the number of channels in the convolution kernel as the input.

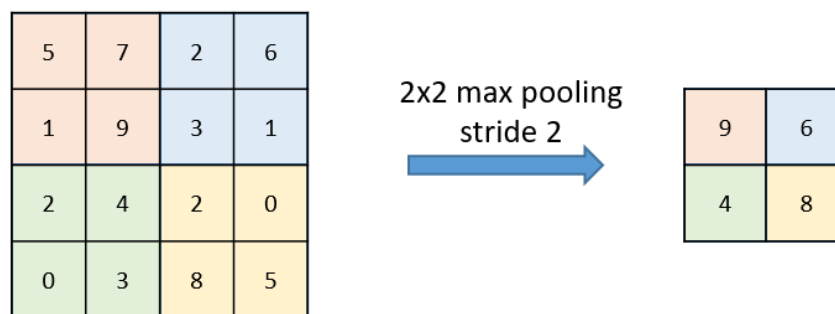


Figure 4: Figure showing an example of a max pooling layer with a kernel size of 2 and stride of 2.

A pooling layer is used to downsample the input feature maps. It takes an input array of shape $d_1 \times d_2 \times n$ and outputs an array of shape $r_1 \times r_2 \times n$. Note that it does **not** change the number of channels, but typically reduces the number of spatial dimensions, i.e., $r_1 < d_1$ and $r_2 < d_2$. In order to do this, we have a **kernel** of shape $k_1 \times k_2$ and a stride s . For each channel, we take either the max or the average of all the points in the window of size $k_1 \times k_2$. Then we slide the window by s pixels and repeat until we have performed this operation over the entire input image. This is illustrated in Figure 4. When the operation performed over each sliding window is max, it is called **max pooling**, whereas when the operation is averaging, then it is called **average pooling**. Using similar notation as above, the function computed by a max pooling layer is

$$Z[r_1, r_2, c] = \text{MaxPool}(X)[r_1, r_2, c] = \max\{X[r_1s : r_1s + k_1 - 1, r_2s : r_2s + k_2 - 1, c]\}.$$

Replacing the max function with the average function, we get an average pooling layer. This function is abstracted away as `pool_fn` in the code. Note that `pool_fn` takes an array as input and outputs a single float. The notation on the right hand side should be read as array slicing as in `numpy`.

Traditional CNNs operate on images by combining convolutional layers with pooling layers to progressively “shrink” the spatial size of the input until it is small enough to be fed to fully-connected network layers for classification.

3 The Neural Nets Package

3.1 Structure

We have provided a modularized codebase for constructing neural networks. The codebase has the following structure.

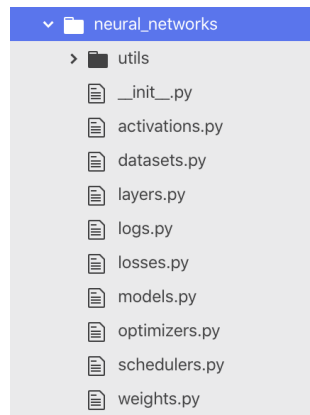


Figure 5: The structure of the starter codebase.

As you can see, the modules in the codebase reflect the structure outlined above. Different losses, activations, layers, optimizers, hyperparameters, and neural network architectures can be combined to yield different architectures.

In the codebase we have provided, each layer is an object with a few relevant attributes.

- **parameters:** An `OrderedDict` containing the weights and biases of the layer.
- **gradients:** An `OrderedDict` containing the gradients of the loss with respect to the weights and biases of the layer, with the same keys as **parameters**.

- **cache**: An `OrderedDict` containing intermediate quantities calculated in the forward pass that are useful for the backward pass.
- **activation**: An `Activation` instance that is the activation function applied by this layer.
- **n_in**: The number of input units (or, input channels in the case of a CNN).
- **n_out**: The number of output units (or, output channels in the case of CNN).

You will pass the layer a parameter that selects an activation function from those defined in `activations.py`. This will be stored as an attribute of the layer, which can be called as `layer.activation()`. The forward and backward passes of the layer are defined by the following methods.

- **forward** This method takes as input the output `X` from the previous layer (or input data). This method computes the function $\phi(\cdot)$ from above, combining the input with the weights `W` and bias `b` that are stored as attributes. It returns an output `out` and saves the intermediate value `Z` to the `cache` attribute, as it is needed to compute gradients in the backward pass.

```
def forward(self, X: np.ndarray) -> np.ndarray:
    """Forward pass: multiply by a weight matrix, add a bias, apply activation.
    Also, store all necessary intermediate results in the `cache` dictionary
    to be able to compute the backward pass.
    """
    ...
```

- **backward** This method takes the gradient of the downstream loss (i.e., the gradient of the loss with respect to the output of the current layer) as input. Then it uses the `cache` from the forward pass to compute the gradient of its output with respect to its inputs and weights. Via chain rule, it then returns the gradient of the loss with respect to the input of the layer.

```
def backward(self, dLdY: np.ndarray) -> np.ndarray:
    """Backward pass for fully connected layer.
    Compute the gradients of the loss with respect to:
    1. the weights of this layer (mutate the `gradients` dictionary)
    2. the bias of this layer (mutate the `gradients` dictionary)
    3. the input of this layer (return this)
    """
    ...
```

Each activation function has a similar (but simpler) structure:

```
class Linear(Activation):
    def __init__(self):
        super().__init__()

    def forward(self, Z: np.ndarray) -> np.ndarray:
        """Forward pass for f(z) = z."""
        return Z

    def backward(self, Z: np.ndarray, dY: np.ndarray) -> np.ndarray:
        """Backward pass for f(z) = z."""
        return dY
```

3.2 Testing

We have provided you with some local test cases that are run using the `unittest` module built into Python for testing. We test the `forward` and `backward` methods of all the layers that you implement, along with a

few that you don't have to implement (which should already be passing their tests). We also test the weight initialization of the FullyConnected layer so that you have some guidance when you implement a layer for the first time. When you are done with the homework, **your code should be passing all the required tests**, i.e. you should have 23/25 tests passing at the end, with the exception to convolution layer and batch normalization's backward pass. You are not required to implement them to get full credit, but it would be great if you can work on that as well.

To run all the tests, make sure that you are in the root directory of the project (and **not** in either of the `neural_networks/` and `tests/` directories). Then run

```
python -m unittest -v
```

A total of 25 tests will be run. Before you implement anything, 17 tests should fail and 8 should pass.

The tests are located in the `tests/` directory. **Please do not modify the pre-existing tests** because they rely on data generated by a correct implementation, and any modifications can cause the tests to fail. The data is located in the `tests/data` directory. However, if you wish to write your own tests in addition to the ones that are already provided, then please refer to [this tutorial](#) and [the official documentation](#). Note that we don't require you to write your own tests as part of this homework, but **the unit tests are not comprehensive, and you should know that implementing the unit tests successfully does not imply that you have a correct implementation**.

The tests are organized into separate classes that mirror the structure of the classes that they test. For example, the `forward` method of the `Conv2D` class located in `neural_networks/layers.py` is tested by the `test_forward` method of the `TestConv2D` class located in `tests/test_layers.py`. If you want to run that specific test, you can use the following command **root directory of the project**.

```
python -m unittest -v tests.test_layers.TestConv2D.test_forward
```

If you instead want to run all the tests defined in the `TestConv2D` class, you should run

```
python -m unittest -v tests.test_layers.TestConv2D
```

And if you want to run all the tests in `test_layers.py`, then you should run

```
python -m unittest -v tests.test_layers
```

3.3 Autograder

Note that **the local tests above are not comprehensive, so it is possible that your code passes all the tests locally but is still broken and gives you poor performance**. They can still be a valuable debugging tool but if you want to properly test for correctness, submit your code to Gradescope by following the instructions on the front page of this PDF. The Gradescope autograder generates a suite of randomized tests (with randomized parameters and inputs) every time it is run and will also test for certain gradients that the local test cases miss.

You are welcome to submit to the autograder as frequently as you wish. However, since the local tests are quicker to run, are fully deterministic and don't require you to submit to Gradescope every time, they should be your first priority as a debugging tool.

3.4 Debugging

We have also included a python notebook named `check_gradients.ipynb`, which you can further use for debugging your layers' gradient computations. This can provide more debuggability than simply running

the provided unit tests. It will compare the gradients you implement for various layers against gradients computed numerically as

$$\frac{\partial f}{\partial x_k}(a) \approx \frac{f(\dots, a_k + \epsilon, \dots) - f(\dots, a_k - \epsilon, \dots)}{2\epsilon}$$

for some small ϵ . If your gradient implementations are correct, they should be close enough to the numerical approximations such that the error between them is very small (usually on the order of 10^{-8} or smaller).

Something to note, however, is that this notebook only checks if the gradients you implement in the backward pass are consistent with the implementation of your forward pass. It's possible to have an incorrect forward pass implementation and a consequently incorrect backward pass implementation that still yields low errors in the notebook. To check if your implementations are correct, you should submit to the Gradescope autograder!

3.5 Submission

Please run

```
python3 generate_submission.py --help
```

for instructions on how to use the script to extract your implementations for the submission.

The script `generate_submission.py` extracts all the code from your implementations and produces either a \LaTeX or markdown file containing your code implementations, when given the flag `--format latex` and `--format markdown` respectively. The generated document contains your functions in separate sections for activation functions, layers, losses, and the model. These can be sections, subsections, or subsubsections depending on whether you supply the flag `--heading_level 1`, `--heading_level 2`, or `--heading_level 3`.

For example, if you want \LaTeX output with each part (activations, layers, losses, and the model) in a different subsection, with the output saved to `submission.tex`, you would run

```
python3 generate_submission.py --format latex --heading_level 2 --output submission.tex
```

whereas if you want markdown output with each part (activations, layers, losses, and the model) in a different subsubsection, with the output saved to `submission.md`, you would run

```
python3 generate_submission.py --format markdown --heading_level 3 --output submission.md
```

We would suggest running these commands to see exactly what they do.

The markdown document will compile by itself, but you would most likely want to create a markdown cell in a Jupyter Notebook and copy-paste the generated markdown into that cell. That should work seamlessly provided you can already compile Jupyter Notebooks into PDFs.

Note that the \LaTeX document will **not** compile by itself. It is meant to generate code that you can then `\input{}` into your \LaTeX document.

Feel free to play around with the script if you want to. Notably, if you change some function which is not in the `student_implementations` list, then you could add that function to the `student_implementations` list to have the script automatically gather your code from that function (but we think that most students will **not** have to do this).

Questions

The background section of the homework has ended. The following sections contain the questions you must complete. For this homework and this homework only, you may assume that the gradient and the partial derivative of a function is interchangeable, aka the expressions $\nabla_w L$ and $\frac{\partial L}{\partial w}$ are the same.

4 Basic Network Layers

In this question you will implement the layers needed for basic classification neural networks. For each part, you will be asked to 1) derive the gradients, 2) write the matching code, 3) pass the tests.

Keep in mind that your solutions to all layers **must operate on minibatches of data** and should not use loops to iterate over the training points in a minibatch. In your code, you should specify whether you are using elementwise multiplication, which is represented by `*`, or if you're using matrix multiplication, which is represented by either `np.dot` or `@`.

4.1 ReLU

You will implement the ReLU activation function in `activations.py`. ReLU is a very common activation function typically used in the hidden layers of a neural network,

$$\text{ReLU}(\gamma) = \begin{cases} 0 & \gamma < 0, \\ \gamma & \text{otherwise.} \end{cases}$$

The activation function is applied elementwise to a vector or matrix input.

Instructions

1. Derive the gradient of the downstream loss with respect to the pre-activation Z of the ReLU. **You must arrive at a solution in terms of $\partial L / \partial Y$, the gradient of the loss w.r.t. the output of ReLU $Y = \text{ReLU}(Z)$, and the batched input Z , i.e., where $Z \in \mathbb{R}^{B \times n}$.** Include your derivation in your writeup.
Hint: you are allowed to use operations like elementwise multiplication and/or division. We recommend the symbol \odot (`\odot` in LaTeX) for elementwise multiplication.
2. Implement the forward and backward passes of the ReLU activation in the script `activations.py`. Include your code in your writeup (either a screenshot or typesetting is fine). **Do not iterate over training examples, use batched operations.**

4.2 Fully-Connected Layer

Now you will implement the forward and backward passes for the fully-connected layer in the `layers.py` script. The code is marked with `YOUR CODE HERE` statements indicating what to implement and where. Please read the docstrings and the function signatures too. Write the fully-connected layer for a general input h that contains a minibatch of B examples with d features.

When implementing a new layer, it is important to manually verify correctness of the forward and backward passes. For debugging tips, please refer back to Section 3.4.

Instructions

1. **Derive $\frac{\partial L}{\partial W}$ and $\frac{\partial L}{\partial b}$** , the gradients of the loss with respect to the weight matrix $W \in \mathbb{R}^{n^{[l]} \times n^{[l+1]}}$ and bias row vector $b \in \mathbb{R}^{1 \times n^{[l+1]}}$ in the fully-connected layer. First derive the gradient element-wise, i.e. find expressions for $\left[\frac{\partial L}{\partial W}\right]_{ij}$ and $\left[\frac{\partial L}{\partial b}\right]_{1i}$, and then stack these elements appropriately to obtain simple vector/matrix expressions for $\frac{\partial L}{\partial W}$ and $\frac{\partial L}{\partial b}$. Repeat this process to also derive the gradient of the loss with respect to the input of the layer $\frac{\partial L}{\partial X}$, which will be passed to lower layers, where $X \in \mathbb{R}^{B \times n^{[l]}}$ is the batched input. **Write your final solution for each of the gradients in terms of $\frac{\partial L}{\partial Z}$** , which you have already obtained in the previous subpart, where $Z = XW + \mathbf{1}b$ and $\mathbf{1} \in \mathbb{R}^{B \times 1}$ is a column of ones. Include your derivations in your writeup.

Note that $\mathbf{1}b$ is a matrix (it's an outer product) whose each row is the row vector b so we are adding the same bias vector to each training point's pre-activation during the forward pass: this is the mathematical equivalent of numpy broadcasting. Include your derivations in your writeup.

2. Implement the forward and backward passes of the fully-connected layer in `layers.py`. First, initialize the weights of the model using `_init_parameters`, which takes the shape of the design matrix as input and initializes the parameters, cache, and gradients of the layer. The backward method takes in an argument `dLdY`, the gradient of the loss with respect to the output of the layer, which is computed by higher layers and backpropagated. This should be incorporated into your gradient calculation. In your writeup, include the code you have implemented. **Do not loop over training points; use batched operations.**

4.3 Softmax Activation

Next, we need to define an activation function for the output layer. The ReLU activation function returns continuous values that are (potentially) unbounded to the right. Since we are building a classifier, we want to return *probabilities* over classes. The softmax function has the desirable property that it outputs a probability distribution. That is, the softmax function squashes continuous values into the range $[0, 1]$ and normalizes the outputs so that they add up to 1. For this reason, many classification neural networks use the softmax activation. The softmax activation takes in a vector a of k un-normalized values a_1, \dots, a_k and outputs a probability distribution over the k possible classes. The forward pass of the softmax activation on input a_i is

$$\hat{y}_i(a) = \frac{e^{a_i}}{\sum_{j=1}^k e^{a_j}},$$

where k ranges over all elements in a . Due to issues of numerical stability, the following modified version of this function is commonly used.

$$\hat{y}_i(a) = \frac{e^{a_i - m}}{\sum_{j=1}^k e^{a_j - m}},$$

where $m = \max_{j=1}^k a_j$. We strongly recommend implementing the latter formula in code (but using the former formula in your derivations).

Instructions

1. Derive the Jacobian of the softmax activation function. **You do not need to use batched inputs for this question; an answer for a single training point is acceptable. You do not need to write out the entire matrix, but please write out what $\partial \hat{y}_i / \partial a_j$ is for an arbitrary (i, j) pair, where $\hat{y} = \text{softmax}(a)$.** Write out the expression in as simple a form as you can, in terms of components of \hat{y} . Include your derivation in your writeup.
2. Implement the forward and backward passes of the softmax activation in `activations.py`. We recommend vectorizing the backward pass for efficiency. For this question **only** may you use a “for” loop over the training points in the minibatch. In your writeup, include the code you have implemented.

4.4 Cross-Entropy Loss

For this classification network, we will be using the multi-class cross-entropy loss function

$$L = -y \cdot \ln \hat{y},$$

where y is the binary one-hot vector encoding the ground truth labels and \hat{y} is the network's output, a vector of probabilities over classes. Note that $\ln \hat{y}$ is \hat{y} with the natural log applied elementwise to it and \cdot represents the dot product between y and $\ln \hat{y}$. The cross-entropy cost calculated for a minibatch of B training points is

$$L = -\frac{1}{B} \sum_{i=1}^B y_i \cdot \ln \hat{y}_i.$$

Let $Y \in \mathbb{R}^{B \times k}$ and $\hat{Y} \in \mathbb{R}^{B \times k}$ be the one-hot labels and network outputs for the B training points, stacked in a matrix. Then, y_i and \hat{y}_i in the expression above are just the i th rows of Y and \hat{Y} .

Instructions

1. Derive $\partial L / \partial \hat{Y}$ the gradient of the cross-entropy cost with respect to the network's predictions, \hat{Y} . **You must use batched inputs.** Include your derivation in your writeup.

Hint: you are allowed to use operations like elementwise multiplication and/or division!

2. Implement the forward and backward passes of the cross-entropy cost. Note that in the codebase we have provided, we use the words “loss” and “cost” interchangeably. This is consistent with most large neural network libraries, though technically “loss” denotes the function computed for a single datapoint whereas “cost” is computed for a batch. You will be computing over minibatches. **Do not iterate over training examples; use batched operations.** In your writeup, include the code you have implemented.

4.5 BatchNorm Layers

In this problem, you will derive some gradients for batch normalization and implement the forward (and optionally backward) pass.

Hint: you are allowed to use operations like elementwise multiplication and/or division. We recommend the symbol \odot (`\odot` in LaTeX) for elementwise multiplication.

1. Derive the gradient of the batch normalization layer. Suppose you have the gradient of the downstream derivative $\frac{\partial L}{\partial z}$. Now derive the gradients
 - (a) $\frac{\partial L}{\partial \beta}$.
 - (b) $\frac{\partial L}{\partial \gamma}$. You may represent this with respect to \hat{h} , which is h after normalization.
 - (c) **(OPTIONAL)** $\frac{\partial L}{\partial h}$. For this part, it would be much easier if you first start with a computational graph, and then break down the derivative that way.
2. Fill in the initialization and forward passes of the `BatchNorm1D` layer in `layers.py`. In your writeup, include the code you have implemented. Remember that when filling in your code for the `forward` function, you should add a keyword `mode`. There are two modes, `train` and `test`, during `test` mode, you'll need to use your running average.

Disclaimer: when you are initializing the layer, you should start by setting the moving averages $\hat{\mu}$ and $\hat{\sigma}_i^2$ to zeros. This is important because otherwise you will fail the unit test and autogradors!
3. **(OPTIONAL)** Fill in the backward passes of the `BatchNorm1D` layer in `layers.py`. Include the code you have implemented as well.

5 Two-Layer Fully Connected Networks

Now, you will use the methods you've written to train a two-layer network (also referred to as a one-*hidden*-layer network). You will use the **Iris Dataset**, which contains 4 features for 3 different classes of irises.

Instructions

1. Fill in the `forward`, `backward`, and `predict` methods for the `NeuralNetwork` class in `models.py`. In your writeup, include the code you have implemented. Define the parameters of your network in `train_ffnn.py`. We have provided you with several other classes that are critical for the training process.
 - The data loader (in `datasets.py`), which is responsible for loading batches of data that will be fed to your model during training. You may alter the data loader to handle data preprocessing if you like. All datasets you are given have not been normalized or standardized.
 - The stochastic gradient descent optimizer (in `optimizers.py`), which performs the gradient updates and optionally incorporates [a momentum term](#).
 - The learning rate scheduler (in `schedulers.py`), which handles the optional learning rate decay. You may choose to use either a constant or exponentially decaying learning rate.
 - Weight initializers (in `weights.py`). We provide you with many options to explore, but we recommend using `xavier_uniform` as a default.
 - A logger (in `logs.py`), which saves hyperparameters and learned parameters and plots the loss as your model trains.

Outputs will be saved to the folder `experiments/`. You can change the name of the folder a given run saves to by changing the parameter called `model_name`. Be careful about overwriting folders; if you forget to change the name and perform a run with identical hyperparameters, your previous run will be overwritten!

2. Train a 2-layer neural network on the Iris Dataset while varying the following hyperparameters.
 - Learning rate
 - Hidden layer size (number of units per hidden layer)

You must try at least 3 different *combinations* of these hyperparameters. Report the results of your exploration, including the values of the parameters you explored and which set of parameters gave the best test error. Provide plots showing the loss versus iterations for your best model and report your final test error.

6 CNN Layers

In this problem, you will write the forward and backward passes for the convolutional and pooling layers that are used within convolutional neural networks.

Again, remember that all your implementations and derived gradients must be for minibatches, and not single examples. For a CNN layer, the input tensor is of the shape (B, H, W, C) where B is the number of minibatch images, H and W are the height and width of each image (the spatial dimensions), and C is the number of channels. (For the first CNN layer, $C = 3$ for RGB images or 1 for grayscale images; for subsequent CNN layers, C is the number of filters applied by the previous layer.)

6.1 The Einsum Function

The `np.einsum` function is a powerful tool for performing various linear algebra operations on arrays in NumPy, a popular numerical computing library in Python. It allows for efficient computation of multi-dimensional array operations through a concise and flexible notation. It is named after the Einstein summation convention.

The `np.einsum` function takes in one or more input arrays and a string specifying the desired output format. The string consists of subscripts that define how the arrays should be multiplied, summed, and contracted. The output of the function is a new array that represents the result of the specified operation.

Here are some basic rules that help you understand how to use `np.einsum`, as this may help you with some of the coding parts in the CNN section.

- Einsum notation uses subscripts to represent the dimensions of the input arrays. Each subscript string is a range of single uppercase or lowercase letters, with each letter denoting an axis of a numpy array. For example, the subscript string `i` represents a 1-D array of length i while the subscript string `ijk` represents an array of shape (i, j, k) .
- The einsum string consists of two or more subscript strings separated by a comma, followed by an arrow (`->`), and a final subscript string that specifies the dimensions of the output array. For example, `i,j->ij` specifies the multiplication of two arrays of lengths i and j , yielding an output array with shape (i, j) . You might recognize this as the outer product between the two input arrays.
- The same subscript letter can appear in multiple input subscript strings, indicating that the corresponding dimensions should be multiplied and summed over. For example, `ij,j->i` performs an elementwise multiplication and sum over the second axis of the first input array and the entire second input array, yielding an output array with one axis and length i . You might recognize this as the regular matrix-vector product between the first input array (a matrix of shape (i, j)) and the second input array (a vector of length j).
- If a subscript appears only in the output subscript string, the corresponding axis is preserved in the output array. For example, `ij->j` returns a one-dimensional array of length j , whose components are the components of the input array summed along the first axis.
- Ellipses (...) can be used to represent a range of axes in an input or output subscript string. For example, `i...->i` performs a summation over all axes except the first of the input array, yielding a one-dimensional output array with length i .

For more details please refer to the NumPy API. This [tutorial](#) should also be very helpful!

Please accomplish the following tasks using both regular NumPy methods and `np.einsum`. Confirm you get the same answer by subtracting the two results and show that its norm is zero (or virtually zero, barring numerical approximation errors). In your writeup, include any code you write as well as a confirmation that both methods yield the same result (for example, printing the norm and including it as a screenshot is fine).

1. For a random matrix $A \in \mathbb{R}^{5 \times 5}$, find its trace.
2. For random matrices $A, B \in \mathbb{R}^{5 \times 5}$, compute their matrix product.
3. For a batch of random matrices of shapes $(3, 4, 5)$ and $(3, 5, 6)$ (the batch size is 3 here), compute their batchwise matrix product (the resulting batch will have shape $(3, 4, 6)$).

6.2 Convolutional Layer

1. Derive the gradient of the loss with respect to the input and parameters (weights and biases) of a convolutional layer. **For this question your answer may be in the form of individual component partial derivatives. Assume you have access to the full 3d array $\frac{\partial L}{\partial Z[d_1, d_2, n]}$, which is the gradient of the pre-activation w.r.t the loss** You may also ignore stride and assume both the filter and image are infinitely zero-padded outside of their bounds. Include your derivations in your writeup.
 - (a) What is $\frac{\partial L}{\partial b[f]}$ for an arbitrary $f \in [1, \dots, n]$?
 - (b) What is $\frac{\partial L}{\partial W[i, k, c, f]}$ for arbitrary i, k, c, f indexes?
 - (c) What is $\frac{\partial L}{\partial X[x, y, c]}$ for arbitrary x, y, c indexes?
2. Fill in the forward passes of the Conv2D layer in `layers.py`. In your writeup, include the code you have implemented. Unlike the mathematical derivation, you will have to account for padding and stride, as well as batching, in your code. **Do not iterate over training examples; use batched operations.** You may find the einsum operation helpful here.
3. **(OPTIONAL)** Fill in the backward passes of the Conv2D layer in `layers.py`. **Do not iterate over training examples; use batched operations.** You may find the einsum operation helpful here.

6.3 Pooling Layers

1. Explain how we can use the backprop algorithm to compute gradients through the max pooling and average pooling operations. (A plain English answer will suffice; equations are optional.) Carefully consider the values you must keep track of!
2. Fill in the forward and backward passes of the `Pool2D` layer in `layers.py`. In your writeup, include the code you have implemented. **Do not iterate over training examples; use batched operations.**

6.4 (Optional) Convolutional Neural Network

Convolutional neural networks take considerably longer to train than the feedforward layers we have built, and the `numpy` implementation you have written will be impractically unoptimized compared to industry-grade neural network packages. So, you are not required to train a proper CNN for this part of the assignment.

That said, if you wish to see your convolutional and pooling layers in action, we have included a file called `train_conv.py` where you can construct a CNN and train it on the **MNIST Dataset**. This can be another good way to test the correctness of your CNN layers (think of this as an integration test, in contrast with the unit tests). The default architecture implemented in that file is called **LeNet** and, along with the default hyperparameters, it should achieve around a 96–97% validation accuracy after just one training epoch, assuming your neural network layers are correctly implemented. Feel free to play around with different hyperparameters to see how they might impact CNN training. You don't have to submit anything for this part in your writeup.

7 PyTorch

As with every homework, you are allowed to use any setup you wish. However, we highly recommend that you use [Google Colab](#) for free access to GPUs, which will significantly improve the speed of neural network training. Instructions are provided in the notebook itself. If you have access to GPUs locally, then feel free to run the notebook on your computer.

The following sections mirror the Colab Notebook and provide the deliverables for each question.

7.1 CNN for Fashion MNIST

Please see the Google Colab Notebook [here](#) that contains the questions.

Deliverables:

1. Provide code for training an CNN on the fashion MNIST dataset (can be tagged from colab notebook and in the code appendix)
2. A plot of the training and validation loss for each epoch of training for at least 8 epochs.
3. A plot of the training and validation accuracy for each epoch, achieving a final validation accuracy of at least 84%.

7.2 Transfer Learning for CIFAR-10

Please see the Google Colab Notebook [here](#) that contains the questions.

Deliverables:

1. Provide the code for training your CNN model (can be in appendix).
2. Submit to Kaggle and include your test accuracy in your report. Our simple reference solution gets a test accuracy of 80% after 3 epochs.
3. Provide at least 1 training curve for your model, depicting loss per epoch or step after training for at least 5 epochs. In addition to validation accuracy after every epoch of training, also include the validation accuracy before training the network in your plot.
4. Briefly explain your network structure and training regime, and how you think your design choices contributed to its performance. Please also explicitly state the number of trainable parameters in your model.
5. Initialize a new network with the same structure as your CNN model, and unfreeze all weights. Now train this network on CIFAR-10 again using your training script for the same number of epochs as your transfer learning model, and compare their performance. Does it give better computational efficiency or learn a more effective model?

Submission Checklist

Please ensure you have completed the following before your final submission.

At the beginning of your writeup...

1. Have you copied and hand-signed the honor code specified in Question 1?
2. Have you listed all students (Names and ID numbers) that you collaborated with?

In your writeup for Question 7...

1. Have you included your **Kaggle Score** and **Kaggle Username**?
2. Have you included your generated plots and visualizations?

At the end of the writeup...

1. Have you provided a code appendix including all code you wrote in solving the homework?

Code Submission

1. Have you dragged and dropped every file in the [neural networks package](#), along with `train_ffnn.py`, into the Gradescope submission box?
2. Have you included a README in your archive containing any special instructions to reproduce your results?
3. Have you included the code you wrote for the PyTorch section with the rest of your submission?

Submissions

1. Have you submitted your written solutions to the Gradescope assignment titled **HW6 Write-Up** and **selected pages** appropriately?
2. Have you submitted your code files to the Gradescope assignment titled **HW6 Code**?
3. Have you submitted your test set predictions for **CIFAR-10** dataset to the appropriate Kaggle challenge?
4. Is your Kaggle submission in integer format? Submissions with decimal points will be assigned a test accuracy of zero by Kaggle.

Congratulations! You have completed Homework 6.