# EECS545 Machine Learning
# Homework #4

**Due Date: Tuesday March 15, 2022 at 11:55 PM**

**Reminder:** This homework is mainly about neural networks and running some scripts might be time-consuming. So we highly recommend you to start working on this homework sooner rather than later. This time, you will work with the **py** files that we provide to you for programming questions. You will be asked to write code for some methods in these **py** files. While you are encouraged to discuss problems in a small group (up to 5 people), you should write your own solutions and source codes independently. In case you worked in a group for the homework, please include the names of your collaborators in the homework submission. Your answers should be as clear and concise as possible. Please post all questions to Piazza with a reference to the specific question in the subject line (E.g. RE: Homework 4, Q1(b)).

**Submission Instruction:** You should submit both **solution** and **source code**. We may inspect your source code submission visually and run the code to check the results. Please be aware that your points may be deducted if you don't follow the instructions listed below:

- Submit **solution** (write-up) to **Gradescope**
  - Your solution should contain your answers (derivations, plots, etc) to **all** questions (typed or hand-written). **Your derivation of the gradient in Q1 should also be included in your written solution**.
  - Your solution should be self-contained and self-explained, regardless of the source code submission.
- Submit **codes** to **Gradescope**. Please do not upload any data files.

In summary, your source code submission for HW4 should be a single zip file which should include all the pre-provided **py** files. Or if you prefer, you can upload individual files to Gradescope directly. The autograder on Gradescope will check if you have all necessary files submitted.

**Source Code Instruction:** Your source code should run under an environment with Python 3.6+. Please install the packages in our starter code so that they could be correctly imported. Please do not use any other libraries or change the directory structure.

Please do **NOT** include the data files `coco_captioning.zip`, `hymenoptera_data.zip`, or `mnist.npz` in your code submission because these are very large files. Use relative path instead of absolute path to load the data. Please note that the outputs of your source code must match with your written solution.

# 1 [25 points] Neural Network Layer Implementation

In this problem, you will get the opportunity to implement various neural network layers. $X, Y$ will represent the input and the output of the layers, respectively. For this problem, we use the row-major notation here (i.e. each row of $X$ and $Y$ corresponds to one data sample) to be compatible with the multi-dimensional array structure of Numpy. Furthermore, $L$ is the scalar valued loss function of $Y$.

For each layer, you should include your derivation of the gradient in your written solution. In addition to the derivations, implement the corresponding forward and backward passes in `layers.py`. More importantly, please ensure that you use vectorization to make your layer implementations as efficient as possible. You should use `autograder.py` to help you in checking the correctness of the your implementation. Note, however, that for each layer, the autograder just checks the result for *one specific example*. Passing these tests does not necessarily mean that your implementation is 100% correct.

**Important Note**: In this question, any indices involved ($i, j$, etc.) in the mathematical notation start from 1, and not 0. In your code, however, you should use 0-based indexing (standard Numpy notation).

(a) **[9 points] Fully-Connected Layer**

In this question, you will be deriving the gradient of a fully-connected layer. For this problem, consider $X \in \mathbb{R}^{N \times D_{\text{in}}}$, where $N$ is the number of samples in a batch. Additionally, consider a dense layer with weight parameters of the form of $W \in \mathbb{R}^{D_{\text{in}} \times D_{\text{out}}}$ and bias parameters of the form of $b \in \mathbb{R}^{1 \times D_{\text{out}}}$.

As we saw in the lecture, we can compute the output of the layer through a simple matrix multiply operation. Concretely, this can be seen as $Y = XW + B$. In this equation, we denote the output matrix as $Y \in \mathbb{R}^{N \times D_{\text{out}}}$ and the bias matrix as $B \in \mathbb{R}^{N \times D_{\text{out}}}$ where each row of $B$ is the vector $b$. (Please note that we are using row-vector notation here to make it compatible with Numpy/PyTorch, but the lecture slides used column-vector notation, which is standard notation for most ML methods. We hope that the distinction is clear from the context.)

Please note, that the matrix multiply operation stated above is generally useful to compute batch outputs (i.e. simultaneously computing the output of $N$ samples in the batch). However, for the purposes of computing the gradient, you might first want to consider the single-sample output operation of this fully-connected layer, which can be stated in row-major notation as $y^{(n)} = x^{(n)}W + b$ where $y^{(n)} \in \mathbb{R}^{1 \times D_{out}}$ and $x^{(n)} \in \mathbb{R}^{1 \times D_{in}}$ for $1 \leq n \leq N$. Here, the index "$(n)$" in the superscript denotes $n$-th example, not the layer index, and $x_i^{(n)} = X_{n,i}$ denotes $i$-th dimension of $n$-th example $x^{(n)}$.

Note: it might be fruitful for you to consider this expression as a summation and then calculate the gradient for individual parameters for a single-example case. After this, you can try to extend it to a vector/matrix format for the involved parameters.

Now, **compute the partial derivatives (in matrix form)** $\frac{\partial L}{\partial W} \in \mathbb{R}^{D_{in} \times D_{out}}$, $\frac{\partial L}{\partial b} \in \mathbb{R}^{1 \times D_{out}}$, $\frac{\partial L}{\partial X} \in \mathbb{R}^{N \times D_{in}}$ **in terms of** $\frac{\partial L}{\partial Y} \in \mathbb{R}^{N \times D_{out}}$. For technical correctness, you might want to start with writing the gradient with non-vectorized form involving $\frac{\partial L}{\partial Y_j^{(n)}} \in \mathbb{R}$, where $\frac{\partial L}{\partial Y_j^{(n)}}$ is the gradient with respect to j-th element of n-th sample in $Y$ with $1 \leq n \leq N$ and $1 \leq j \leq D_{\text{out}}$. In addition, you might find the Kronecker Delta useful in this derivation.

**Hint for $\frac{\partial L}{\partial W}$**

Please note that we use a "matrix/vector" notation $\frac{\partial L}{\partial W}$ to denote a Jacobian matrix in $\mathbb{R}^{D_{\text{in}} \times D_{\text{out}}}$ where the element in $i$-th row and $j$-th column is $\frac{\partial L}{\partial W_{i,j}}$ and $W_{i,j}$ is the $i$-th row, $j$-th column element of $W$ with $1 \leq i \leq D_{\text{in}}$ and $1 \leq j \leq D_{\text{out}}$. Here, you would probably want to calculate the gradient $\frac{\partial L}{\partial W_{i,j}}$ using the formula $\frac{\partial L}{\partial W_{i,j}} = \sum_{n=1}^{N} \sum_{m=1}^{D_{out}} \frac{\partial L}{\partial Y_m^{(n)}} \frac{\partial Y_m^{(n)}}{\partial W_{i,j}}$, and then vectorize $\frac{\partial L}{\partial W_{i,j}}$ to get $\frac{\partial L}{\partial W}$.

**Hint for $\frac{\partial L}{\partial b}$**

Please note that $\frac{\partial L}{\partial b}$ is a vector in $\mathbb{R}^{1\times D_{out}}$. Ideally, you might want to start by using the formula $\frac{\partial L}{\partial b_j} = \sum_{n=1}^{N}\sum_{m=1}^{D_{out}} \frac{\partial L}{\partial Y_m^{(n)}} \frac{\partial Y_m^{(n)}}{\partial b_j}$ and then move on to derive $\frac{\partial L}{\partial b}$.

**Hint for $\frac{\partial L}{\partial X}$**

Please note that $\frac{\partial L}{\partial X}$ is a matrix in $\mathbb{R}^{N\times D_{in}}$. In order to derive this gradient, you can start by using the formula $\frac{\partial L}{\partial X_i^{(n)}} = \sum_{n'=1}^{N}\sum_{m=1}^{D_{out}} \frac{\partial L}{\partial Y_m^{(n')}} \frac{\partial Y_m^{(n')}}{\partial X_i^{(n)}}$ Following this, you can say that $\frac{\partial L}{\partial X_i^{(n)}} = \sum_{m=1}^{D_{out}} \frac{\partial L}{\partial Y_m^{(n)}} \frac{\partial Y_m^{(n)}}{\partial X_i^{(n)}}$ because the $n$-th sample in $X$ is only related with the $n$-th sample in $Y$ and $\frac{Y_m^{(n')}}{\partial X_i^{(n)}}$=0 for all $n' \neq n$.

(b) **[5 points] ReLU**

Let $X \in \mathbb{R}^{N\times D}$ be a 2-D array and $Y = \text{ReLU}(X)$. In this case, ReLU is applied to $X$ in an element-wise fashion. For an element $x \in X$, the corresponding output is $y = \text{ReLU}(x) = \max(0, x)$. You can observe that $Y$ will have the same shape as $X$. **Express $\frac{\partial L}{\partial X}$ in terms of $\frac{\partial L}{\partial Y}$**, where $\frac{\partial L}{\partial X}$ has the same shape as $X$. Now, implement the `relu_forward` and `relu_backward` methods in `layers.py`.

**Hint**: you may need to use the element-wise product to express $\frac{\partial L}{\partial X}$

(c) **[11 points] Convolution**

For better visualization of the convolution operations, you can use the following link: `https://bit.ly/3H0Rjhn`. Consider the following 2D arrays/tensors: $\mathbf{a} \in \mathbb{R}^{H_a\times W_a}$ and $\mathbf{b} \in \mathbb{R}^{H_b\times W_b}$. Note, $\mathbf{a}$ is the image and $\mathbf{b}$ is the filter with $H_a > H_b$ and $W_a > W_b$.

**Valid convolution:** In this problem, we can define the valid convolution as follows:

$$(\mathbf{a} *_{\text{valid}} \mathbf{b})_{i,j} = \sum_{m=i}^{i+H_b-1}\sum_{n=j}^{j+W_b-1} a_{m,n} b_{i-m+H_b,j-n+W_b}$$

Here, $1 \le i \le H_a - H_b + 1$ and $1 \le j \le W_a - W_b + 1$. Please note that the convolution operation we discussed in class is **valid convolution**, and does not involve any zero padding. This operation produces an output of size $(H_a - H_b + 1) \times (W_a - W_b + 1)$.

**Filtering:** Moreover, it might also be useful to consider the **filtering** operation $*_{\text{filt}}$, defined by:

$$(\mathbf{a} *_{\text{filt}} \mathbf{b})_{i,j} = \sum_{m=i}^{i+H_b-1}\sum_{n=j}^{j+W_b-1} a_{m,n} b_{m-i+1,n-j+1} = \sum_{p=1}^{H_b}\sum_{q=1}^{W_b} a_{i+p-1,j+q-1} b_{p,q}$$

Here, $1 \le i \le H_a - H_b + 1$ and $1 \le j \le W_a - W_b + 1$. Please note that the filtering operation generates an output of size $(H_a - H_b + 1) \times (W_a - W_b + 1)$. In summary, the filtering operation is similar to the valid convolution, except that the filter is not flipped when computing the weighted sum. You can think of filtering as an inner product between the $H_b \times W_b$ sized kernel and image patch of the same size inside the input image where we don't flip the kernel.

**Full convolution:** Finally, for deriving the gradient through convolution layer, the full convolution operation may be useful. This operation is defined as follows:

$$\boxed{(\mathbf{a} *_{\text{full}} \mathbf{b})_{i,j} = \sum_{m=i-H_b+1}^{i} \sum_{n=j-W_b+1}^{j} a_{m,n} b_{i-m+1,j-n+1}}$$

Here, $1 \leq i \leq H_a+H_b-1$ and $1 \leq j \leq W_a+W_b-1$. The full convolution can be thought of as zero padding $\mathbf{a}$ on all sides with one less than the size of the kernel, and then performing valid convolution using the modified input tensor $\mathbf{a}$. Concretely, this means that we will pad $\mathbf{a}$ by $H_b-1$ rows on the top and bottom, followed by $W_b-1$ columns on the left and right. In the definition of full convolution, $a_{m,n} = 0$ if $m < 1$ or $n < 1$ or $m > H_a$ or $n > W_a$. This operation produces an output of size $(H_a+H_b-1) \times (W_a+W_b-1)$.

Here, assume the input to the layer to be $X \in \mathbb{R}^{N \times C \times H \times W}$, where $N$ is the number of images in the batch, $C$ is the number of channels, $H$ is the height of the image, $W$ is the width of the image. Furthermore, consider a convolutional kernel $K \in \mathbb{R}^{F \times C \times H' \times W'}$, where $F$ represents the number of filters present in this layer. The output of this layer is given by $Y \in \mathbb{R}^{N \times F \times H'' \times W''}$ where we have $H'' = H - H' + 1$ and $W'' = W - W' + 1$.

**Backpropagation through the convolution layer**: We now consider the output of the layer $Y_{n,f}$ which can be defined as $Y_{n,f} = \sum_{c=1}^{C} X_{n,c} *_{\text{valid}} \overline{K}_{f,c}$. In this case, $\overline{K}_{f,c}$ represents the flipped filter, which can be more concretely defined for each element as $\overline{K}_{f,c,i,j} = K_{f,c,H'+1-i,W'+1-j}$.

This means that the output expression $Y_{n,f} = \sum_{c=1}^{C} X_{n,c} *_{\text{valid}} \overline{K}_{f,c}$ can also be represented by the expression $Y_{n,f} = \sum_{c=1}^{C} X_{n,c} *_{\text{filt}} K_{f,c}$. Therefore, you can implement $Y_{n,f} = \sum_{c=1}^{C} X_{n,c} *_{\text{filt}} K_{f,c}$ in the `conv_forward` function in the `layers.py` script.

Finally, considering the upstream gradient to be denoted by $\frac{\partial L}{\partial Y_{n,f}}$, **show that**:

$$\frac{\partial L}{\partial X_{n,c}} = \sum_{f=1}^{F} K_{f,c} *_{\text{full}} \left( \frac{\partial L}{\partial Y_{n,f}} \right)$$

and

$$\frac{\partial L}{\partial K_{f,c}} = \sum_{n=1}^{N} X_{n,c} *_{\text{filt}} \left( \frac{\partial L}{\partial Y_{n,f}} \right)$$

Additionally, implement the `conv_forward` and `conv_backward` methods in `layers.py`.

**Hint for $\frac{\partial L}{\partial X_{n,c}}$**

Please note that the gradient $\frac{\partial L}{\partial X_{n,c}} \in \mathbb{R}^{H \times W}$, where the $(i,j)^{th}$ element of $\frac{\partial L}{\partial X_{n,c}}$ is represented as $\frac{\partial L}{\partial X_{n,c,i,j}}$. Theoretically, this expression measures the change induced in loss $L$ when you perturb the input element $X_{n,c,i,j}$ which is the pixel value in the $i$-th row, $j$-th column, and $c$-th channel of the $n$-th sample image in the batch.

In order to derive this gradient expression, start with the following expression:

$$Y_{n,f} = \sum_{c=1}^{C} X_{n,c} *_{\text{filt}} K_{f,c}$$

Now, use the definition of the $*_{\text{filt}}$ operation given in the problem statement (the double summation expression) to represent $Y_{n,f}$ as a triple summation (in this hint, we assume the inner summation

variables to be $p$ and $q$). Using this triple summation expression, derive the following partial derivative:

$$\frac{\partial Y_{n,f,p,q}}{\partial X_{n,c,i,j}}$$

This should yield a relatively simple expression. Following this, you should use the following chain rule expression:

$$\frac{\partial L}{\partial X_{n,c,i,j}} = \sum_{f=1}^{F} \sum_{p=1}^{H''} \sum_{q=1}^{W''} \frac{\partial Y_{n,f,p,q}}{\partial X_{n,c,i,j}} \frac{\partial L}{\partial Y_{n,f,p,q}}$$

Observe that you already have the value of $\frac{\partial Y_{n,f,p,q}}{\partial X_{n,c,i,j}}$, so this can be substituted into the expression for $\frac{\partial L}{\partial X_{n,c,i,j}}$. Finally, try to use a change of variables in the inner two summations so that the inner two summations can be used to represent the **full convolution** operation. This should yield the required expression.

**Hint for $\frac{\partial L}{\partial K_{f,c}}$**
Use the same steps as in the previous hint. Please note that you might not need to do a change of summation variables here, and that you should finally try to represent the inner two summations as the **filtering** operation. Additionally, for the chain rule step, please use the following expression:

$$\frac{\partial L}{\partial K_{f,c,i,j}} = \sum_{n=1}^{N} \sum_{p=1}^{H''} \sum_{q=1}^{W''} \frac{\partial Y_{n,f,p,q}}{\partial K_{f,c,i,j}} \frac{\partial L}{\partial Y_{n,f,p,q}}$$

This should help you arrive at the final expression.

### Implementation note for questions 2-4

To solve question 2-4, please read through `solver.py` and familiarize yourself with the API. To build the models, please use the intermediate layers you implemented in `layers.py`. After doing so, use a `Solver` instance to train the models. Additionally, **please vectorize your code to ensure that your networks can train in a reasonable amount of time.**

## 2 [20 points] Multi-class classification with Softmax

(a) Implement softmax loss layer `softmax_loss` in `layers.py`. This function outputs the softmax loss averaged over all input examples.

(b) Implement softmax multi-class classification using the starter code provided in `softmax.py`.

(c) Train a two layer neural network with a softmax loss function on the MNIST dataset using the script `digit_classification.py`. Identify and report the best setting for the number of hidden units based on the performance on validation set. Please ensure that you try atleast 6 different settings for the number of hidden units. Using the optimal setting from the previous step, train your network again, and report the accuracy obtained on the test set.

## 3 [20 points] Convolutional Neural Network for multi-class classification

(a) Implement the forward and backward passes of the max-pooling layer in `layers.py`

(b) Implement CNN for softmax multi-class classification using the starter code provided in `cnn.py`.

(c) Train the CNN multi-class classifier on MNIST dataset with `digit_classification.py`. Using the hyperparameters specified in the solver instance, train the CNN on the MNIST dataset and report the accuracy obtained on the test set. [Tip: to make the computation more efficient, please consider vectorizing the implementation of convolution and max-pool layer.]

# 4 [20 points] Application to Image Captioning

In this problem, you will apply the RNN module that has been partially implemented to build an image captioning model. The link to download data is provided in the comment in `image_captioning.py`. Please unzip `coco_captioning.zip` to get the data.

(a) At every timestep we use a fully-connected layer to transform the RNN hidden vector at that timestep into scores for each word in the vocabulary. This is very similar to the fully-connected layer that you implemented in Q1. Implement the forward pass in `temporal_fc_forward` function and the backward pass in `temporal_fc_backward` function in `rnn_layers.py`. [Tip: `autograder.py` could also help debugging.]

(b) Now that you have the necessary layers in `rnn_layers.py`, you can combine them to build an image captioning model. Implement the forward and backward pass of the model in the `loss` function for the RNN model in `rnn.py`.

(c) With the RNN code now ready, run the script `image_captioning.py` to get learning curves of training loss and the caption samples. Report the learning curves and the caption samples based on your well-trained network.

# 5 [15 points] Transfer Learning

By working on this problem, you will be more familiar with PyTorch and can run experiments for two major transfer learning scenarios. The link to download data is provided in the comment in `transfer_learning.py`. Please unzip `hymenoptera_data.zip` to get the data.

(a) Fill in the code in the `train_model` function which is a general function for model training.

(b) Fill in the code in the `visualize_model` function to briefly visualize how the trained model performs on validation images.

(c) Fill in the code in the `finetune` function. Instead of random initialization, we initialize the network with a pre-trained network. Rest of the training looks as usual.

(d) Fill in the code in the `freeze` function. We will freeze the weights for all of the network except that of the final fully connected layer. This last fully connected layer is replaced with a new one with random weights and only this layer is trained.

(e) Run the script and report the accuracy on validation dataset for these two scenarios.

# Credits

Some questions adopted/adapted from http://cs231n.stanford.edu/.