EECS 442 Computer Vision: Homework 4

Instructions

- This homework is **due at 11:59:59 p.m. on Monday March 29th, 2021**.
- The submission includes two parts:

  1. **To Canvas:** submit a `zip` file of all of your code.
     We have indicated questions where you have to do something in code in red.
     We have indicated questions that will be autograded in purple.
     Your zip file should contain a single directory which has the same name as your uniqname. If I (Justin, uniqname `justincj`) were submitting my code, the zip file should contain a single folder `justincj/` containing all required files.
     What should I submit? At the end of the homework, there is a canvas submission checklist provided. We provide a script that validates the submission format here. If we don't ask you for it, you don't need to submit it; while you should clean up the directory, don't panic about having an extra file or two.
  2. **To Gradescope:** submit a `pdf` file as your write-up, including your answers to all the questions and key choices you made.
     We have indicated questions where you have to do something in the report in blue.
     You might like to combine several files to make a submission. Here is an example online link for combining multiple PDF files: https://combinepdf.com/.
     The write-up must be an electronic version. **No handwriting, including plotting questions.** LaTeXis recommended but not mandatory.

Python Environment We are using Python 3.7 for this course. You can find references for the Python standard library here: https://docs.python.org/3.7/library/index.html. To make your life easier, we **recommend** you to install Anaconda for Python 3.7.x (https://www.anaconda.com/download/). This is a Python package manager that includes most of the modules you need for this course.

We will make use of the following packages extensively in this course:

- Numpy (https://docs.scipy.org/doc/numpy-dev/user/quickstart.html)
- SciPy (https://scipy.org/)
- Matplotlib (http://matplotlib.org/users/pyplot_tutorial.html)

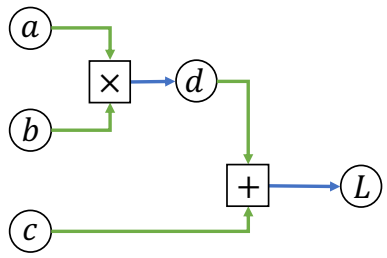
1 Computational Graphs and Backprop [25 points]

We have seen that representing mathematical expressions as *computational graphs* allows us to easily compute gradients using backpropagation. After writing a mathematical expression as a computational graph, we can easily translate it into code. In this problem you'll gain some experience with backpropagation in a simplified setting where all of the inputs, outputs, and intermediate values are all scalar values instead vectors, matrices, or tensors.

In the *forward pass* we receive the inputs (leaf nodes) of the graph and compute the output. The output is typically a scalar value representing the loss L on a minibatch of training data.

In the *backward pass* we compute the derivative of the graph's output L with respect to each input of the graph. There is no need to reason *globally* about the derivative of the expression represented by the graph; instead when using backpropagation we need only think *locally* about how derivatives flow backward through each node of the graph. Specifically, during backpropagation a node that computes $y = f(x_1, \ldots, x_N)$ receive an *upstream gradient* $\frac{\partial L}{\partial y}$ giving the derivative of the loss with respect the the node output and computes *downstream gradients* $\frac{\partial L}{\partial x_1}, \ldots, \frac{\partial L}{\partial x_N}$ giving the derivative of the loss with respect to the node inputs.

Here's an example of a simple computational graph and the corresponding code for the forward and backward passes. Notice how each **outgoing edge** from an operator gives rise to one line of code in the forward pass, and each **ingoing edge** to an operator gives rise to one line of code in the backward pass.

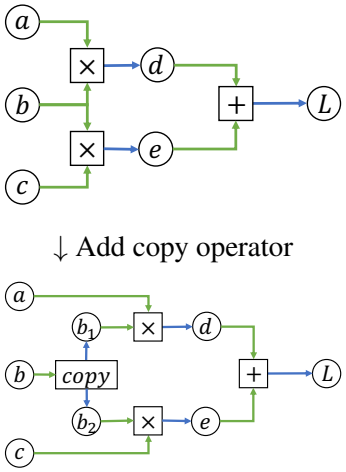



```python
def f(a, b, c):
  d = a * b        # Start forward pass
  L = c + d

  grad_L = 1.0     # Start backward pass
  grad_c = grad_L
  grad_d = grad_L
  grad_a = grad_d * b
  grad_b = grad_d * a

  return L, (grad_a, grad_b, grad_c)
```

Sometimes you'll see computational graphs where one piece of data is used as input to multiple operations. In such cases you can make the logic in the backward pass cleaner by rewriting the graph to include an explicit `copy` operator that returns multiple copies of its input. In the backward pass you can then compute separate gradients for the two copies, which will sum when backpropagating through the copy operator:




```python
def f(a, b, c):
  b1 = b           # Start forward pass
  b2 = b
  d = a * b1
  e = c * b2
  L = d + e

  grad_L = 1.0     # Start backward pass
  grad_d = grad_L
  grad_e = grad_L
  grad_a = grad_d * b1
  grad_b1 = grad_d * a
  grad_c = grad_e * b2
  grad_b2 = grad_e * c
  grad_b = grad_b1 + grad_b2  # Sum grads for copies

  return L, (grad_a, grad_b, grad_c)
```

Task 1: Implementing Computational Graphs [15 points]

Below we've drawn three computational graphs for you to practice implementing forward and backward passes. The file `backprop/functions.py` contains stubs for each of these computational graphs. You can use the driver program `backprop/backprop.py` to check your implementation.

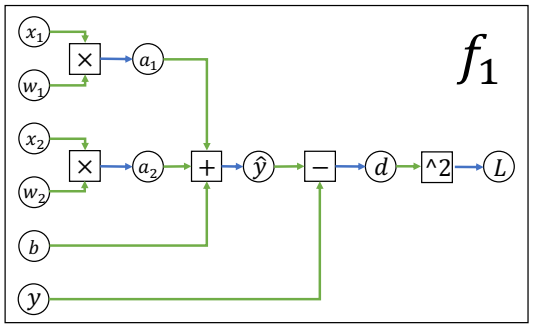
Implement the forward and backward passes for the three computational graphs below.

The file `backprop/backprop-data.pkl` contains sample inputs and outputs for the three computational graphs; the driver program loads inputs from this file for you when checking your forward passes.

To check the backward passes, the driver program implements *numeric gradient checking*. Given a function $f : \mathbb{R} \to \mathbb{R}$, we can approximate the gradient of f at a point $x_0 \in \mathbb{R}$ as:

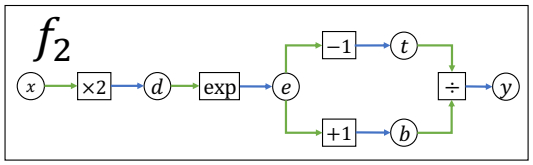
$$\frac{\partial f}{\partial x}(x_0) \approx \frac{f(x_0 + h) - f(x_0 - h)}{2h}$$

Each of these computational graphs implements a function or operation commonly used in machine learning. Can you guess what they are? (This is just for fun, not required).



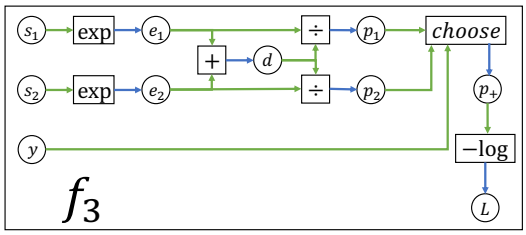
The subtraction node computes $d = \hat{y} - y$

The `^2` node computes $L = d^2$



The $\times 2$ node computes $d = 2x$

The $\div$ node computes $y = t/b$



y is an integer equal to either 1 or 2.
You don't need to compute a gradient for y.

The $\div$ nodes compute $p_1 = e_1/d$ and $p_2 = e_2/d$.

The `choose` node outputs outputs p_1 if $y = 1$, and outputs p_2 if $y = 2$.

Task 2: Write your own computational graph [10 points]

In your report, draw a computational graph for any function of your choosing. It should have at least five operators. (You can hand-draw the graph and include a picture of it in your report.)

In the file `backprop/functions.py`, implement a forward and backward pass through your computational graph in the function `f4`. You can modify the function to take any number of input arguments. After implementing `f4`, you can use the driver script to perform numeric gradient checking. Depending on the functions in your graph, you may see errors $\geq 10^{-8}$ even with a correct backward pass. This is ok!

2 Optimization and Fitting [25 points]

In this problem you will solve the same problem as HW3 Task 3: Fitting an affine transform to a set of correspondences. However rather than fitting the transformation via RANSAC as you did in HW3, in this problem you will solve the same problem using backpropagation and gradient descent.

More concretely, suppose we have a set of N 2D correspondences $p_i \leftrightarrow p'_i$ where $p_i = (x_i, y_i) \in \mathbb{R}^2$ and $p'_i = (x'_i, y'_i) \in \mathbb{R}^2$. We want to find an affine transform parameterized by $S \in \mathbb{R}^{2\times 2}, t \in \mathbb{R}^2$ such that

$$p'_i = \begin{bmatrix} x'_i \\ y'_i \end{bmatrix} \approx \begin{bmatrix} S_{1,1} & S_{1,2} \\ S_{2,1} & S_{2,2} \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} + \begin{bmatrix} t_1 \\ t_2 \end{bmatrix} = Sp_i + t$$

by solving the optimization problem

$$\arg\min_{S,t} \frac{1}{N} \sum_{i=1}^{N} \|Sp_i + t - p'_i\|^2$$

via gradient descent on S and t.

Task 3: Fitting an Affine Transform [15 points]

We have provided a driver script `fitting/main.py` for this problem.

(a) **Implement the function `affine_transform_loss`** in the file `fitting/fitting.py`. We have provided code for you that implements numeric gradient checking; just run `python main.py gradcheck`.

(b) **Implement the function `fit_affine_transform`** in the file `fitting/fitting.py` to minimize the loss function defined above using gradient descent. You can run this fitting routine on the provided data with the command `python main.py fit`.

(c) Experiment with the learning rate and number of steps for gradient descent in order to find settings that give a good fit to the provided data. You can provide your own settings for these hyperparameters by passing `--learning-rate` and `--steps` command-line flags to `python main.py fit`. You should be able to achieve a final loss less than 10^{-4} with a total runtime of less than 5 seconds. For your best result, create a plot of the SGD losses with the flag `--loss-plot` and an animated GIF of the fitting process with the flag `--animated-gif`.
In your report, report your best (S, t), your hyperparameters, and include the loss plot.

Task 4: Gradient Descent vs RANSAC [10 points]

Discuss in your report: Between HW3 and HW4, you have now fit affine transforms using two different algorithms: RANSAC and Gradient Descent. What are some advantages and disadvantages of each algorithm? You should describe this in about three or four sentences.

3 Fully-Connected Neural Networks [50 points]

In this question you will implement and train a fully-connected neural network to classify images.

For this question you cannot use any deep learning libraries such as PyTorch or TensorFlow.

Task 5: Modular Backprop API [20 points]

In the previous questions on this assignment you used backpropagation to compute gradients by implementing monolithic functions that combine the forward and backward passes for an entire graph. As we've discussed in lecture, this monolithic approach to backpropagation isn't very modular – if you want to change some component of your graph (new loss fucntion, different activation function, etc) then you need to write a new function from scratch.

Rather than using monolithic backpropagation implementations, most modern deep learning frameworks use a *modular API* for backpropagation. Each primitive operator that will be used in a computational graph implements a *forward* function that computes the operator's output from its inputs, and a *backward* function that receives upstream gradients and computes downstream gradients. Deep learning libraries like PyTorch or TensorFlow provide many predefined operators with corresponding forward and backward functions.

To gain experience with this modular approach to backpropagation, you will implement your own miniature modular deep learning framework. The file `neuralnet/layers.py` defines forward and backward functions for several common operators that we'll need to implement our own neural networks.

Each forward function receives one or more numpy arrays as input, and returns: (1) A numpy array giving the output of the operator, and (2) a *cache* object containing values that will be needed during the backward pass. The backward function receives a numpy array of upstream gradients along with the cache object, and must compute and return downstream gradients for each of the inputs passed to the forward function.

Along with forward and backward functions for operators to be used in the middle of a computational graph, we also define functions for *loss functions* that will be used to compute the final output from a graph. These loss functions receive an input and return both the loss and the gradient of the loss with respect to the input.

This modular API allows us to implement our operators and loss functions once, and reuse them in different computational graphs. For example, we can implement a full forward and backward pass to compute the loss and gradients for linear regression in just a few lines of code:

```
from layers import fc_forward, fc_backward, l2_loss

def linear_regression_step(X, y, W, b):
    y_pred, cache = fc_forward(X, W, b)
    loss, grad_y_pred = l2_loss(y_pred, y)
    grad_X, grad_W, grad_b = fc_backward(grad_y_pred, cache)
    return grad_W, grad_b
```

In the file `neuralnet/layers.py` you need to complete the implementation of the following:

(a) **Fully-connected layer**: `fc_forward` and `fc_backward`.

(b) **ReLU nonlinearity**: `relu_forward` and `relu_backward` which applies the function $ReLU(x_i) = \max(0, x)$ elementwise to its input.

(c) **Softmax Loss Function**: `softmax_loss`. The softmax loss function receives a matrix $x \in \mathbb{R}^{N \times C}$ giving a batch of classification scores for N elements, where for each element we have a score for each of C different categories. The softmax loss function first converts the scores into a set of N probability distributions over the elements, defined as:

$$p_{i,c} = \frac{\exp(x_{i,c})}{\sum_{j=1}^{C} \exp(x_{i,j})}$$

The output of the softmax loss is then given by

$$L = -\frac{1}{N} \sum_{i=1}^{N} \log(p_{i,y_i})$$

where $y_i \in \{1, \ldots, C\}$ is the ground-truth label for the ith element.

A naïve implementation of the softmax loss can suffer from *numeric instability*. More specifically, large values in x can cause overflow when computing exp. To avoid this, we can instead compute the softmax probabilities as:

$$p_{i,c} = \frac{\exp(x_{i,c} - M_i)}{\sum_{j=1}^{C} \exp(x_{i,j} - M_i)}$$

where $M_i = \max_c x_{i,c}$. This ensures that all values we exponentiate are < 0, avoiding any potential overflow. It's not hard to see that these two formulations are equivalent, since

$$\frac{\exp(x_{i,c} - M_i)}{\sum_{j=1}^{C} \exp(x_{i,j} - M_i)} = \frac{\exp(x_{i,c}) \exp(-M_i)}{\sum_{j=1}^{C} \exp(x_{i,j}) \exp(-M_i)} = \frac{\exp(x_{i,c})}{\sum_{i=1}^{C} \exp(x_{i,j})}$$

Your softmax implementation should use this max-subtraction trick for numeric stability. You can run the script `neuralnet/check_softmax_stability.py` to check the numeric stability of your softmax loss implementation.

(d) **L2 Regularization**: `l2_regularization` which implements the L2 regularization loss

$$L(W) = \frac{\lambda}{2} \|W\|^2 = \frac{\lambda}{2} \sum_i W_i^2$$

where the sum ranges over all scalar elements of the weight matrix W and λ is a hyperparameter controlling the regularization strength.

After implementing all functions above, you can use the script `neuralnet/gradcheck_layers.py` to perform numeric gradient checking on your implementations. The difference between all numeric and analytic gradients should be less than 10^{-9}.

Keep in mind that numeric gradient checking does not check whether you've correctly implemented the forward pass; it only checks whether the backward pass you've implemented actually computes the gradient of the forward pass that you implemented.

Task 6: Implement a Two-Layer Network [10 points]

Your next task is to implement a two-layer fully-connected neural network using the modular forward and backward functions that you just implemented.

In addition to using a modular API for individual layers, we will also adopt a modular API for classification models as well. This will allow us to implement multiple different types of image classification models, but train and test them all with the same training logic.

The file `neuralnet/classifier.py` defines a base class for image classification models. You don't need to implement anything in this file, but you should read through it to familiarize yourself with the API. In order to define your own type of image classification model, you'll need to define a subclass of `Classifier` that implements the `parameters`, `forward`, and `backward` methods.

In the file `neuralnet/linear_classifier.py` we've implemented a `LinearClassifier` class that subclasses `Classifier` and implements a linear classification model using the modular layer API from the previous task together with the modular classifier API. Again, you don't need to implement anything in this file but you should read through it to get a sense for how to implement your own classifiers.

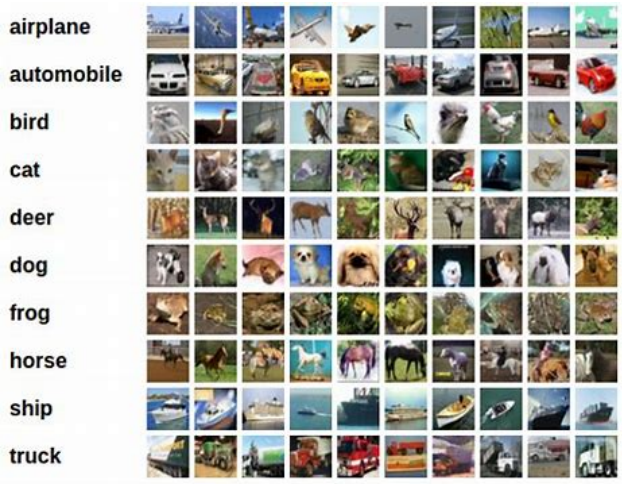
Now it's your turn! In the file `neuralnet/two_layer_net.py` we've provided the start to an implementation of a `TwoLayerNet` class that implements a two-layer neural network (with ReLU nonlinearity).

Complete the implementation of the `TwoLayerNet` class. Your implementations for the `forward` and `backward` methods should use the modular forward and backward functions that you implemented in the previous task.

After completing your implementation, you can run the script `gradcheck_classifier.py` to perform numeric gradient checking on both the linear classifier we've implemented for you as well as the two-layer network you've just implemented. You should see errors less than 10^{-10} for the gradients of all parameters.

Task 7: Training Two-Layer Networks [20 points]

You will train a two-layer network to perform image classification on the CIFAR-10 dataset. This dataset consists of 32×32 RGB images of 10 different categories. It provides 50,000 training images and 10,000 test images. Here are a few example images from the dataset:



You can use the script `nerualnet/download_cifar.sh` to download and unpack the CIFAR10 dataset.

The file `neuralnet/train.py` implements a training loop. We've already implemented a lot of the logic here for you. You don't need to do anything with the following files, but you can look through them to see how they work:

- `neuralnet/data.py` provides a function to load and preprocess the CIFAR10 dataset, as well as a `DataSampler` object for iterating over the dataset in minibatches.
- `neuralnet/optim.py` defines an `Optimizer` interface for objects that implement optimization algorithms, and implements a subclass `SGD` which implements basic stochastic gradient descent with a constant learning rate.

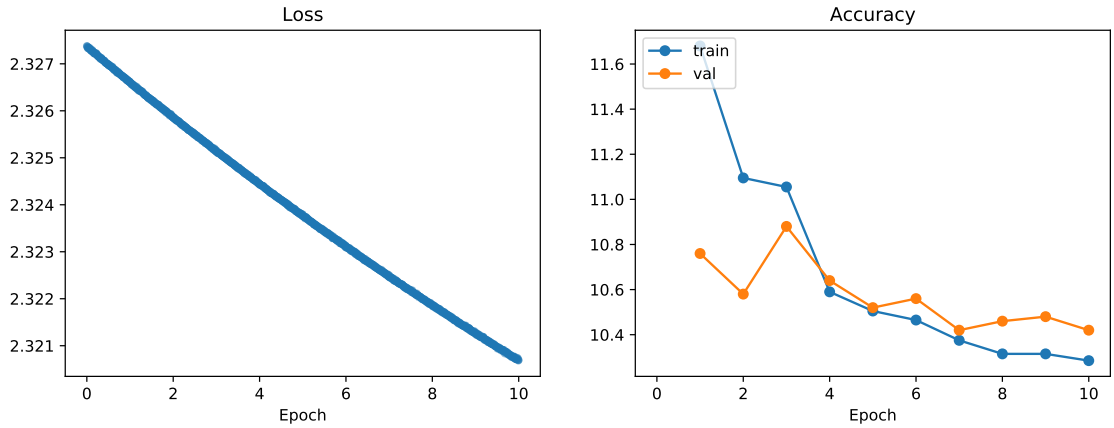
Implement the `training_step` function in the file `neuralnet/train.py`.

This function inputs the model, a minibatch of data, and the regularization strength; it computes a forward and backward pass through the model and returns both the loss and the gradient of the loss with respect to the model parameters. The loss should be the sum of two terms:

(a) A *data loss* term, which is the softmax loss between the model's predicted scores and the ground-truth image labels

(b) A *regularization loss* term, which penalizes the L2 norm of the weight matrices of all the fully-connected layers of the model. You should not apply L2 regularization to the biases.

Now it's time to train your model! Run the script `neuralnet/train.py` to train a two-layer network on the CIFAR-10 dataset. The script will print out training losses and train and val set accuracies as it trains. After training concludes, the script will also mke a plot of the training losses as well as the training and validation-set accuracies of the model during training; by default this will be saved in a file `plot.pdf`, but this can be customized with the flag `--plot-file`. You should see a plot that looks like this:



Unfortunately, it seems that your model is not training very effectively – the training loss has not decreased much from its initial value of ≈ 2.3, and the training and validation accuracies are very close to 10% which is what we would expect from a model that randomly guesses a category label for each input.

You will need to tune the hyperparameters of your model in order to improve it. Try changing the hyperparameters of the model in the provided space of the `main` function of `neuralnet/train.py`. You can consider changing any of the following hyperparameters:

- `num_train`: The number of images to use for training
- `hidden_dim`: The width of the hidden layer of the model
- `batch_size`: The number of examples to use in each minibatch during SGD
- `num_epochs`: How long to train the model. An *epoch* is a single pass through the training set.
- `learning_rate`: The learning rate to use for SGD
- `reg`: The strength of the L2 regularization term

You should tune the hyperparameters and train a model that achieves at least 40% on the validation set. After tuning your model, run your best model **exactly once** on the test set using the script `neuralnet/test.py`.

In your report, include the loss / accuracy plot for your best model, describe the hyperparameter settings you used, and give the final test-set performance of your model.

You may not need to change all of the hyperparameters; some are fine at their default values. Your model shouldn't take an excessive amount of time to train. For reference, our hyperparameter settings achieve $\approx 45\%$ accuracy on the validation set in ≈ 5 minutes of training on a 2019 MacBook Pro.

To gain more experience with hyperparameters, you should also tune the hyperparameters to find a setting that results in an *overfit model* that achieves $\geq 75\%$ accuracy on the *training set*.

In your report, include the loss / accuracy plot for your overfit model and describe the hyperparameter settings you used.

As above, this should not take an excessive amount of training time – we are able to train an overfit model that achieves $\approx 80\%$ accuracy on the training set within about a minute of training.

HINT: It's easier to overfit a smaller training set.

Task 8: Implementing More Features (OPTIONAL)

Ordinarily we'd ask you to implement a few more neural net features in this homework, but this year we've decided to trim the assignment down a bit.

However if you want to go further in this assignment, we have a few ideas for optional extensions that you can try to implement on your own:

- Implement a generalized multi-layer network that can have an arbitrary number of hidden layers. You can define this as a new subclass of `Classifier`.
- Implement other activation functions, such as sigmoid, tanh, Exponential Linear Units (ELU). You can implement these as additional forward / backward functions in `neuralnet/layers.py`.
- Implement other optimization algorithms, such as SGD + Momentum. You can implement these as a new subclass of `Optimizer` in `neuralnet/optim.py`.

If you implement these or any other features, let us know in your report! You can include loss and accuracy curves to show how well your new features do compared to the model you trained in Task 7.

Canvas Submission Checklist

In the `zip` file you submit to Canvas, the directory named after your uniqname should include the following files:

- `functions.py`
- `fitting.py`
- `layers.py`
- `train.py`
- `two_layer_net.py`

The following are **optional**

- `optim.py`