AI535 Deep Learning – Homework 2

Feed-Forward Neural Networks

Overview and Objectives. In this homework, we are going to implement a simple feed-forward neural network for image classification from scratch. This should concertize concepts like linear layers, ReLU, back-propagation, batch stochastic gradient descent, and ADAM.

How to Do This Assignment. All questions that require a response will be in blue-gray task boxes. We prefer solutions typeset in LaTeX but will accept scanned written work if it is legible. If a TA can't read your work, they can't give you credit. Submit your solutions to Canvas as a PDF and your code as a `.py` file.

Advice. Start early. The provided skeleton code provides a framework to work in. If your code is painfully slow, you are likely looping when a purely linear algebra solution exists.

1 Written Exercises: Getting to Know Softmax [5pts]

The softmax function is a very common component in deep learning models for classification. Given a real-valued vector $s \in \mathbb{R}^d$, the ith output of the softmax is defined as

$$p_i = \frac{e^{s_i}}{\sum_{j=1}^{d} e^{s_j}}. \tag{1}$$

It is easy to see that all $0 \leq p_i \leq 1$ and $\sum_j p_j = 1$ such that the softmax transforms a real-valued vector to a discrete probability distribution of the same dimension.

1.1 Numerical Stability of Softmax

Computing the softmax on a machine with finite precision runs into some trouble due to overflow or underflow. If one of the s_j's is sufficiently large and positive, both the numerator and denominator may overflow to infinity. Conversely, if all the s_j's are sufficiently large and negative then both the numerator and denominator may underflow to zero. In either case, we no longer get a valid output for softmax. What can we do about this?

▶ `Q1 Stabilized Softmax [1pt].` A common trick to stabilize the softmax is to subtract the maximum score in $\mathbf{s}$. Let s_{max} be the maximum value of $\mathbf{s}$ and show that the following equality holds:

$$p_i = \frac{e^{s_i}}{\sum_j e^{s_j}} = \frac{e^{s_i - s_{max}}}{\sum_j e^{s_j - s_{max}}} \tag{2}$$

Explain how this avoids the problems with underflow and overflow described in the section above.

1.2 Gradients of Softmax and Cross Entropy Loss.

We've stabilized the forward-pass of the softmax function, now let's have a look at the backward pass.

▶ Q2 Gradient of Softmax [2pt]. Derive the gradient of the softmax with respect to its inputs $\frac{\partial \mathbf{p}}{\partial \mathbf{s}}$. Your final expression should be in terms of $\mathbf{p}$.

Often, the probabilities produced by the softmax are the network's output for classification tasks. A very common classification loss paired with the softmax is cross entropy. For a target distribution y and a model-produced distribution p, the cross entropy loss is computed as:

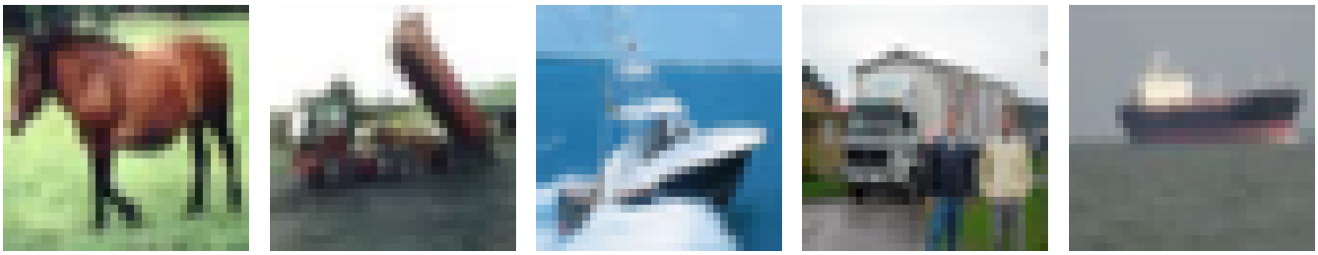
$$CE(y, p) = -\sum_i y_i \log p_i \tag{3}$$

The partial derivative of CE with respect to p_i is $\frac{\partial CE}{\partial p_i} = -\frac{y_i}{p_i}$ which is numerically unstable for the probabilities very near 0 that are commonly produced via softmax. Instead, the cross entropy and softmax functions are typically combined such that their gradient $\frac{\partial CE}{\partial \mathbf{s}}$ is computed directly rather than through chain rule.

▶ Q3 Gradient of Cross Entropy on Softmax [2pt]. Let $\mathbf{p} = softmax(\mathbf{s})$. Derive the gradient of the cross entropy loss between $\mathbf{y}$ and $\mathbf{p}$ with respect to $\mathbf{s}$ – i.e. $\frac{\partial CE(y, softmax(\mathbf{s}))}{\partial \mathbf{s}}$.

2 Implementing a Neural Network For Image Classification [20pts]

Simple CIFAR10. In this section, we will implement a feed-forward neural network model for predicting whether an image contains a horse, a ship, or a truck. This is a subset of the CIFAR10 dataset. A few examples from the dataset are shown below:



Each digit is a 32×32 color image with values ranging from 0 to 255. We represent an image as a row vector $x \in \mathbb{R}^{1\times 3072}$ where the image has been serialized into one long vector. Each class has an associated label from 0,1,2 corresponding to its value (0=horse, 1=ship, 2=truck). We provide three dataset splits for this homework – a training set containing 13000 examples, a validation set containing 2000, and our test set containing 3000. Classes are approximately uniform in each split.

2.1 Cross-Entropy Loss for Multiclass Classification

We have a dataset $D = \{\mathbf{x}_i, y_i\}_{i=1}^N$ with $y_i \in \{0, 1, 2\}$. Assume we have a model $f(x; \theta)$ parameterized by a set of parameters θ that predicts $P(Y|X = x)$ (a distribution over our labels given an input). Let's refer to $P(Y = c|X = x)$ predicted from this model as $p_{c|x}$ for compactness. We can write the cross entropy loss as

$$\mathcal{L}(\theta) = -\sum_{i=1}^{N} \log p_{y_i|x_i} \tag{4}$$

In training, we will minimize this by following a batch stochastic approximation of $\nabla_\theta \mathcal{L}$ in gradient descent.

2.2 Implementing Backpropagation for Feed-forward Neural Network

We'll consider feed-forward neural networks composed of a sequence of linear layers $\mathbf{x}W_1 + \mathbf{b}_1$ and ReLU activation functions $ReLU(\cdot)$. As such, a network with 3 of these layers stacked together could be written

$$f(\mathbf{x}) = \mathbf{b}_3 + ReLU(\mathbf{b}_2 + ReLU(\mathbf{b}_1 + \mathbf{x}W_1)W_2)W_3 \tag{5}$$

Note how this is a series of nested functions, reflecting the sequential feed-forward nature of the computation. To make our notation more useful, let's expand this out and name intermediate results

$$\begin{aligned}
\mathbf{z}_1 &= \mathbf{x}W_1 + \mathbf{b}_1 && (6)\\
\mathbf{a}_1 &= ReLU(\mathbf{z}_1) && (7)\\
\mathbf{z}_2 &= \mathbf{a}_1 W_2 + \mathbf{b}_2 && (8)\\
\mathbf{a}_2 &= ReLU(\mathbf{z}_2) && (9)\\
\mathbf{z}_3 &= \mathbf{a}_2 W_3 + \mathbf{b}_3 && (10)
\end{aligned}$$

where $\mathbf{z}$'s are intermediate outputs from the linear layers and $\mathbf{a}$'s are post-activation function outputs. In the case of our dataset, $\mathbf{z}_3$ will have 3 dimensions – a score for each of the possible labels. Finally, the output vector $\mathbf{z}_3$ is not yet a probability distribution so we apply the softmax function such that

$$p_{j|x} = \frac{e^{\mathbf{z}_{3_j}}}{\sum_{c=0}^{2} e^{\mathbf{z}_{3_c}}} \tag{11}$$

and let $\mathbf{p}_{\cdot|x}$ be the vector of these predicted probability values.

Operating on Batches. In practice, we will want to operate on batches of B examples at once for efficiency. The linear layers above could be modified to compute $Z = XW + \mathbf{b}$ for $X \in \mathbb{R}^{B \times input\_dim}$ and $Z \in \mathbb{R}^{B \times output\_dim}$ and we can call this a batched operation. It is straightforward to change the forward pass to operate on these all at once by taking advantage of matrix multiplication and the broadcasted addition that numpy provides. On the backward pass, we simply need to aggregate the gradient of the loss of each data point with respect to our parameters. For example,

$$\frac{\delta L}{\delta W_1} = \sum_{i=1}^{n} \frac{\delta L_i}{\delta W_1} \tag{12}$$

where L_i is the loss of the i'th datapoint and L is the overall loss.

Modular Layers and Backpropagation. As discussed in class and in the first homework, we can consider this network as a computational graph. Here we will work at the level of abstraction of individual layers – linear layers, ReLU layers, and a unified Softmax Cross Entropy layer.

Each layer is responsible for computing its forward and backward computations. Taking the example below of a Sigmoid activation layer, we can see a forward compute that applies the sigmoid element-wise and a backward computation that returns $\partial L/\partial input = \partial L/\partial output * \partial output/\partial input$.

```
class Sigmoid:

  # Given the input, apply the sigmoid function
  # store the output value for use in the backwards pass
  def forward(self, input):
    self.act = 1/(1+np.exp(-input))
    return self.act

  # Compute the gradient of the output with respect to the input
  # self.act*(1-self.act) and then multiply by the loss gradient with
  # respect to the output to produce the loss gradient with respect to the
    input
  def backward(self, grad):
    return grad * self.act * (1-self.act)

  # The Sigmoid has no parameters so nothing to do during a gradient descent
    step
  def step(self,step_size):
    return
```

The Sigmoid layer does not have any parameters. Layers that do have parameters will also need to compute $\partial L/\partial parameter = \partial L/\partial output * \partial output/\partial parameter$ in the backward pass and store it for use in the optimization step later (e.g. a gradient descent update).

▶ Q4 Implement Initialization, Forward, and Backward for Layers [6pts]. The skeleton code provides stub classes for `LinearLayer`, `ReLU`, and `CrossEntropySoftmax`. Finish `__init__`, `forward`, and `backward` functions in each. These should operate on a batch of inputs at a time and should be agnostic to what layers come before or after them. The skeleton code for the linear layer is shown below as an example. We will fill out the optimization `step` function later.

```
class LinearLayer:
  # Initialize weights/bias and make any storage variables for gradients
  def __init__(self, input_dim, output_dim):
    #TODO

  # Compute and return XW+b
  def forward(self, input):
    #TODO

  # Given dL/doutput as grad, compute dL/dparameters and store them,
  # then return dL/dinput so it can be passed backwards
  def backward(self, grad):
    #TODO

  # Take a step of optimization
  def step(self,step_size):
    #TODO
```

Tips:

- Do as much as possible with matrix operations, loops are the enemy of efficiency in python. See `numpy`'s broadcasted addition rules when adding the bais vector for the linear layer.
- Many operations apply element-wise in numpy – for example `np.exp(X)` exponentiates each element of its input matrix X.
- When working out the gradients, consider the single example case first (i.e. not batched). Once you've derived the per-example Jacobian, you can generalize to the batched setting. For gradients with respect to parameters, you'll simply sum up these per-example results. Often this summation can be efficiently done in matrix operations rather than an explicit sum.
- Feel free to store results from the forward pass to make the backward pass easier – see `self.act` sigmoid code above for example.

Feedforward Network. Putting these layers together, we get a fully-connected network. The basic sketch of the network is already provided in the code and is configurable to have a specified number of layers of fixed width. This code is replicated below. You are welcome to modify this to your own custom structure when you are tuning your network later on. Some points to notice is how the `forward`, `backward`, and `step` functions operate. During the forward pass, the output of each layer (as computed by the layer `forward` call) is passed to the layer above it. In the backward pass, the gradient of the loss with respect to the input of each layer (as computed by the layer `backward` call) is passed to the layer before it.

During the optimization `step`, the network just calls individual layer `step`'s. In our case, the only layers with parameters are the `LinearLayer`, but we call all the steps for generality.

```
class FeedForwardNeuralNetwork:

  # Builds a network of linear layers separated by non-linear activations
  def __init__(self, input_dim,output_dim,hidden_dim,num_layers):

    if num_layers == 1: #Just a linear mapping from input to output

      self.layers = [LinearLayer(input_dim, output_dim)]

    else: # At least two layers

      #Layer to map input to hidden dimension size
      self.layers = [LinearLayer(input_dim, hidden_dim)]
      self.layers.append(ReLU())

      # Hidden layers
      for i in range(num_layers-2):
        self.layers.append(LinearLayer(hidden_dim, hidden_dim))
        self.layers.append(ReLU())

      # Layer to map hidden dimension to output size
      self.layers.append(LinearLayer(hidden_dim, output_dim))

  # Given an input, call the forward function of each of our layers
  def forward(self, X):
    for layer in self.layers:
      X = layer.forward(X)
    return X

  # Given an gradient with respect to the network output, call
  # the backward function of each of our layers.
  def backward(self, grad):
    for layer in reversed(self.layers):
      grad = layer.backward(grad)

  # Update each layer's parameters
  def step(self, step_size=0.001):
    for layer in self.layers:
      layer.step(step_size)
```

Optimizing Parameter with ADAM and Weight Decay. At this point in the assignment, your network should be able to calculate gradients for all of its parameters. Now, we need to update our parameters using those gradients. We will use the ADAM optimizer described in class.

Denoting parameters are w, ADAM keeps track of a running average of the gradient and its raw second moment as below where β is a hyperparameter:

$$m^{(t)} = \beta m^{(t-1)} + (1-\beta)\nabla L(w^{(t)}) \tag{13}$$
$$v^{(t)} = \beta v^{(t-1)} + (1-\beta)(\nabla L(w^{(t)}))^2 \tag{14}$$

Then the parameters are updated according to

$$w^{t+1} = w^{(t)} - \alpha \frac{m^{(t)}/(1-\beta)}{\sqrt{v^{(t)}/(1-\beta)} + \epsilon} \tag{15}$$

where the $(1-\beta)$ factors are bias correction terms from the running average.

We also discussed weight decay in class as a means to regularize neural networks. To apply this to our problem, we can simply add a factor of our parameters to our gradients before performing the running average in ADAM. For example, modifying the above equations to:

$$m^{(t)} = \beta m^{(t-1)} + (1-\beta)(\nabla L(w^{(t)}) + \lambda w^{(t)}) \tag{16}$$
$$v^{(t)} = \beta v^{(t-1)} + (1-\beta)(\nabla L(w^{(t)}) + \lambda w^{(t)})^2 \tag{17}$$

where λ is the weight decay hyperparameter. This addition encourages our weights to head towards zero. Note that weight decay is typically not applied to biases in linear layers.

▶ Q5 Implement ADAM and Weight Decay. [4pts] Complete the `step` function in the linear layer by implementing ADAM with weight decay as described in the example above. You will need to introduce class-level variables to maintain the running averages.

The Training Loop. Now that we have forward/backward and optimization handles, let's start actually training. The typical structure for a standard training loop is in pseudo-code below:

```
#for each epoch:
    # scramble data order

    #for each batch in data:
        # compute forward of batch including evaluating the loss

        # compute backward of batch

        # step optimizer

        # bookkeeping to track how loss/accuracy change during training

    # evaluate validation set
```

▶ Q6 Implement Training and Evaluation. [5pts] Implement a training loop in the `main` function of the skeleton code. It should follow the above pseudo-code. Further, it should produce the data used to plot training and validation curves – see the skeleton code for full details.

As a supporting function to your main loop, finish the implementation of the `evaluate` function.

```
def evaluate(model, X_val, Y_val, batch_size):
 # TODO evaluate loss and accuracy for all points in X_val
 return avg_loss, accuracy
```

This should be used to evaluate the validation set in the main training loop and the test set after.

▶ Q7 Tune for Performance. [5pts] Adjust the structure of your model, change initialization, modify data preprocessing, and adjust optimization parameters to achieve strong results. Correctly implemented solutions should easily reach 80% accuracy on the validation set and transfer well to test. To calibrate your efforts, my best-tuned version hits a bit above 85% on test.

Your report should include your training plot generated by the code, final test accuracy, and a written discussion of what you found most impactful while tuning.

Tips:

- It is usually a good idea to focus on *overfitting* first with neural networks as proof the network is working properly. Nearly 100% training accuracy should be possible with small networks.
- Normalizing the input data makes optimization easier. Consider computing the mean and stdev across train then normalizing the train, val, and test splits with them.
- If you want to do data augmentation, see the `displayExample` utility function for an example how to turn the vectors back into images.
- It is worth looking back at the Network Optimization (I) lecture for tips/tricks.

3 Debriefing (required in your report)

1. Approximately how many hours did you spend on this assignment?
2. Would you rate it as easy, moderate, or difficult?
3. Did you work on it mostly alone or did you discuss the problems with others?
4. How deeply do you feel you understand the material it covers (0%–100%)?
5. Any other comments?