CS577: Assignment 2

Jane Downer
Department of Computer Science
Illinois Institute of Technology

October 6, 2022

Abstract
Implementing basic forward and backward passes by hand, in Python, and in the context of computation graphs.

Programming Questions: Task 1

1 Problem Statement

- Implement a three-layer neural network in the Keras/Tensorflow framework. Train on a dataset from the UCI Machine Learning Repository.

2 Proposed Solution

- Build a model suitable for multi-class classification from Keras/Tensorflow layers. Fit model to chosen dataset, using custom step functions and `tf.GradientTape()` to update parameters and track metrics. Use `tf.keras.metrics.categorical_accuracy and tf.keras.metrics.categorical_crossentropy` to assess performance.

3 Implementation Details

- General instructions and notes:
  - At the top of the .ipynb file, change 'ROOT_PATH' to the location where you want to save the final model to (or load the trained model from). If you download and unzip the `'Dry Bean Dataset'` with `!wget` and `!unzip`, this will be where the files are saved. The `winequality-red.csv` data can be loaded directly into the notebook as Pandas dataframes.
  - Program can be run consecutively up until 'Question 2' header. Model will be saved at ROOT_PATH.
  - Use last line of code before 'Question 2' to load existing trained model from ROOT_PATH.
  - Note: as mentioned in the notebook, many methods for this task were borrowed from lecture and from *Deep Learning with Python* [1].
- `load_data(data, test_p, label_column_idx):`
  - Identify categorical column of dataset with `label_column_idx`. Only consider the data falling under three distinct categories – in this case, I chose the first three unique labels appearing in the dataset.
  - Split the data into train and test sets, using `test_p` to indicate the proportion used test set. (I used `test_set = 0.2`, so the model would train on the majority of data while still having a significant and representative test subset to evaluate performance on.)
  - Provide categorical encoding of label data using the `sklearn` library's `OneHotEncoder` module – this converts the labels from strings to binary values which allows us to measure accuracy and loss.
- `build_Model(X_trn, input_size_L1, input_size_L2):`
  - Build three-layer `keras.Model`, using `'sigmoid'` activation for hidden layers and `'softmax'` activation for output layer, as instructed. Treat the input size of the hidden layers as hyperparameters and adjust as needed to improve model performance.
- `BatchGenerator` class:
  - Create a generator class for creating and retrieving batches of training data.
  - Initialize instance with training data, batch size, and number of batches. Use `next()` method to retrieve subsequent batches of dataset during training.
- `fit(model, X_trn, y_trn, epochs, batch_size, optimizer, loss_tracker, acc_tracker, val_loss_tracker, val_acc_tracker):`
  - Use `sklearn` library's `train_test_split` function to further split training data into a portion used only for training and a portion used for validation. I used 20% of the training input for validation, which gave the model a significant portion to train on while leaving an ample amount for testing its ability to generalize.
  - Make best initial guess for number of epochs, batch size, optimizer, and learning rate. Treat these as hyperparameters and try different combinations if necessary after evaluating model performance.

- Start every epoch by resetting loss and accuracy trackers so we can look at the performance of each epoch individually.
- During each epoch, instantiate a training batch generator, and subsequently call each unique training batch (the number of times determined by the size of the data and the specified batch size).
  - For each batch, call `train_step(model, X_batch, y_batch, optimizer, loss_tracker, acc_tracker)` – whose arguments are defined when we call `fit()`.
    - This function passes each batch of data through the current iteration of the model to get a prediction, and computes the loss and accuracy each time. It uses `tf.GradientTape()` to track the gradients of the model parameters, and then updates the training loss tracker and training accuracy tracker each time it is called.
    - Returns log of categorical cross entropy and accuracy for the given batch.
- During each epoch, following `train_step()`, also repeat a similar procedure for the validation data – creating a validation batch generator and running the function `val_step()`.
  - This function is different than `train_step()` in that it does not use `tf.GradientTape()` to compute and track model gradients, and it does not use the optimizer to perform updates – this is because the validation data is used purely for evaluating model performance rather than making changes to the model itself.
- Print metrics for each epoch. Return the trained model along with list of loss and accuracy values (for both training and validation data) for each epoch of training.

I trained the model on the '[Dry Bean Dataset]' from the UCI Machine Learning Repository. Below are the hyperparameter values used in final model:

Train data shape	Test data shape	Validation data shape	Learning Rate	Optimizer	Epochs	Batch size	# Units, hidden layer 1	# Units, hidden layer 2
(4348, 16)	(1360, 16)	(1088,16)	0.001	Adam	200	512	32	16

I chose the Adam optimizer because it is generally fast to converge, and it presented no issues with this task. Learning rate was initially 0.0001, which caused loss to decrease roughly linearly, which suggested that it could be higher. I ended up increasing batch size from my initial guess because training accuracy was increasing rapidly while validation accuracy was slow to improve, and this helped with overfitting. Using 32 and 16 units for the first and second hidden layers (respectively) with this combination of hyperparameter values resulted in a reasonable rate of training that struck a balance between training and validation performance. Lastly, 200 epochs was around the right length of time to capture the learning trends.

Other hyperparameter choices were provided in the instructions – 3 layers, the first two of which used sigmoid activation, and the last of which used softmax activation. Sigmoid provided nonlinearity in the hidden layers of the network, and softmax is appropriate for the output layer for multi-class classification tasks, where it produces results that can be interpreted as the probability of an observation belonging to each possible class. The instructions also specified that we use categorical cross entropy for the loss function, which is appropriate for multi-class classification tasks because it can be used to measure the difference between two probability distributions (in this case, the predicted probability of each labels versus the actual presence of each label).

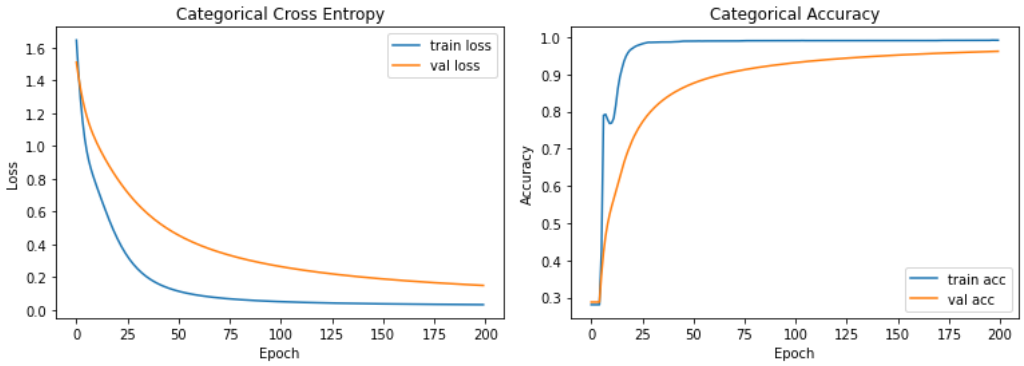
4 Results and Discussion

Below is the training output, which shows the accuracy and categorical cross-entropy at each epoch for both the training and validation data.

```
train: (4348, 16)
val: (1088, 16)
Epoch 0/200             train_loss: 1.645 , val_loss: 1.510 , train_acc: 0.281 , val_acc: 0.288
Epoch 10/200            train_loss: 0.751 , val_loss: 1.018 , train_acc: 0.769 , val_acc: 0.545
Epoch 20/200            train_loss: 0.436 , val_loss: 0.802 , train_acc: 0.969 , val_acc: 0.722
Epoch 30/200            train_loss: 0.250 , val_loss: 0.646 , train_acc: 0.986 , val_acc: 0.806
Epoch 40/200            train_loss: 0.160 , val_loss: 0.535 , train_acc: 0.987 , val_acc: 0.849
Epoch 50/200            train_loss: 0.115 , val_loss: 0.456 , train_acc: 0.989 , val_acc: 0.876
Epoch 60/200            train_loss: 0.090 , val_loss: 0.398 , train_acc: 0.989 , val_acc: 0.894
Epoch 70/200            train_loss: 0.074 , val_loss: 0.353 , train_acc: 0.990 , val_acc: 0.907
Epoch 80/200            train_loss: 0.064 , val_loss: 0.318 , train_acc: 0.990 , val_acc: 0.917
Epoch 90/200            train_loss: 0.057 , val_loss: 0.289 , train_acc: 0.990 , val_acc: 0.925
Epoch 100/200           train_loss: 0.051 , val_loss: 0.265 , train_acc: 0.990 , val_acc: 0.931
Epoch 110/200           train_loss: 0.047 , val_loss: 0.245 , train_acc: 0.990 , val_acc: 0.937
Epoch 120/200           train_loss: 0.044 , val_loss: 0.228 , train_acc: 0.990 , val_acc: 0.941
```

```
Epoch 130/200          train_loss: 0.042 , val_loss: 0.214 , train_acc: 0.990 , val_acc: 0.945
Epoch 140/200          train_loss: 0.040 , val_loss: 0.201 , train_acc: 0.991 , val_acc: 0.949
Epoch 150/200          train_loss: 0.038 , val_loss: 0.190 , train_acc: 0.990 , val_acc: 0.952
Epoch 160/200          train_loss: 0.037 , val_loss: 0.180 , train_acc: 0.991 , val_acc: 0.954
Epoch 170/200          train_loss: 0.036 , val_loss: 0.171 , train_acc: 0.991 , val_acc: 0.957
Epoch 180/200          train_loss: 0.035 , val_loss: 0.163 , train_acc: 0.991 , val_acc: 0.959
Epoch 190/200          train_loss: 0.034 , val_loss: 0.156 , train_acc: 0.991 , val_acc: 0.960
```

The plots below provide visual representations of these values:



Here is the summarized performance of the model on the training and validation data, in addition to the performance on the test set:

```
Final categorical cross-entropy:
    train:    0.033
    test:     0.037
    val:      0.151
Final categorical accuracy:
    train:    0.992
    test:     0.988
    val:      0.962
```

The above results show that the model adequately learned the features of the training set and was able to generalize to unseen data.

Programming Questions: Task 2

1 Problem Statement

- For this task, I implemented a computation graph in Python without use of any built-in deep learning frameworks.

2 Proposed Solution

- Create a class for each relevant type of node. The forward pass of each node transforms the input to that node into a value that is passes to the next node in the graph. The backward pass outputs `upstream_grad*local_grad(parameter)` for each parameter at that given node.
- Create a Computation_Graph class built from instances of the previously-defined node classes. Includes forward() and backward() method that chain together the forward and backward passes for each node. Also includes a train() method for fitting a computation graph model to user-provided data, and a test() method for evaluating the trained model on a set of unseen data. Train() methods uses an implementation of stochastic gradient descent for updating model during training.

3 Implementation Details

- General instructions and notes:
  - None in particular. There are issues with the code that I address at the end of the notebook, as well as in the "Results and Discussion" section of this document.
- Each node class has two methods:
  - `forward()`: processes the input to the given node to be passed to the next node in the graph
  - `backward()`: using upstream gradients and the parameters at the current node to compute local gradients and implement the chain rule
- `Multiply()`: multiplying the input and weights for each layer

    - `forward()`: outputs the product of two input matrices (weights, inputs)
    - `backward()`: for a parameter in a Multiply() node, the output is equivalent to 1 times the value of the other parameter. (i.e., `dX = 1*W`, and `dW = 1*X`).
- `Add()`: adding the bias to the above product
    - `forward()`: outputs the sum of two input matrices (Wx, bias)
    - `backward()`: for a parameter in an Add() node, the output is equivalent to the upstream gradient.
- `Sigmoid()`:
    - `forward()`:
        - Applies the sigmoid function to the input to a given node: $Sigmoid(x) = \frac{1}{1+\exp(-x)}$
        - Provides nonlinear activation for the `Wx+b` term in the given layer
    - `backward()`: outputs the upstream gradient times `sigmoid(a)*(1-sigmoid(a))` for each parameter `a`.
- `Softmax()`:
    - `forward()`:
        - Applies the softmax function to the input to the final layer: $Softmax(z_j) = \frac{\exp(z_j)}{\sum_{i=1}^{k}\exp(z_i)}$
        - Computes the probabilities of an observation belonging to each possible category
    - `backward()`: Outputs the upstream gradient times $softmax(z_j) \cdot \left(\delta_{ij} - softmax(z_i)\right)$ for parameter z_i.
- `CategoricalCrossEntropy()`:
    - `forward()`:
        - Measures the distance between the predicted labels and actual labels
    - `backward()`: outputs $\left(-\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}\right)$ times the upstream gradient given the actual and target labels.
- `Computation_Graph` class:
    - The base architecture includes the three-layer structure described in instructions.
    - `forward(self, X, y, n_units_L1, n_units_L2, W_L1, W_L2, W_L3)`:
        - `X` and `y` define the input data and ground truth labels.
        - `n_units_L1` and `n_units_L1` define the number of nodes in the hidden layers.
        - `W_L1, W_L2, W_L3` define the weight matrices.
            - For the first iteration, unless explicitly provided, these are set to small, random values. In subsequent passes, the `forward()` method receives explicitly-defined weight matrices that have been updated with stochastic gradient descent after the previous iteration (more later).
            - First rows of weight matrices represent bias. Before being fed into the first layer, the weight matrices are split into the weight and bias portions – the weight portions are multiplied by inputs in the `Multiply()` node of each layer, and the bias terms are added to the result in the subsequent `Add()` nodes.
        - Feed data through network, measure loss with categorical cross-entropy function, and use 'softmax' activation in final output layer to determine probability of observation belonging to each of 3 possible classes.
        - Return predictions (`self.y_hat`) and loss (`self.loss`) for the given iteration.
    - `backward(self)`:
        - Receives the output from the computation graph's `forward()` pass as input. Chains together the `backward()` methods for all nodes in the graph, starting with the final node (Categorical cross entropy) and moving backward to the first node (Multiply() in the first layer).
        - Returns the gradients of the weights and biases.
    - `train(self, X_train, y_train, lr, epochs, decay_factor, batch_size)`:
        - `X_train`, `y_train`: user-provided training observations and corresponding labels
        - `batch_size`: user-specified value – determines frequency of training updates.
        - Use `split_into_batches()` function to split `X_train` and `y_train` into batches of user-specified size.
            - `split_into_batches(M, batch_size)`:
                - Takes array `M` and returns as list of `int(M.shape[0]/batch_size)` batches
        - For each batch:
            - Convert Feed batch forward through computation graph to generate predictions and loss.
            - Compute accuracy with `OneHotAccuracy(y_batch, y_hat)`, which computes the percentage of that `y_hat` assigns the highest probability to the ground_truth category (given by `y_batch`).
            - Append epoch loss and accuracy to respective lists to keep track of model performances.

- Update weights and biases with stochastic gradient descent.
    - `SGD(parameter, gradient, lr, decay_factor)`:
        - `parameter`: parameter to update
        - `gradient`: the derivative of the loss with respect to the given parameter
        - `lr`: user-specified learning rate
        - `decay_factor`: user-specified value to scale the learning-rate during an update
        - Update formula:
            - `parameter = parameter – lr*gradient*decay_factor`
    - Returns per-epoch loss and accuracy values.
- `Test(self, X_test, y_test):`
    - Takes user-provided test observations and their ground-truth labels.
    - Generates predictions and loss with `forward()` method of the `Computation_Graph` class.
    - Calculates accuracy with `OneHotaccuracy()` function described above.
    - Returns loss and accuracy.
- Backpropagate and make updates to model.

4 Results and Discussion

- As mentioned in the notebook, I ran into problems when it came to executing the code for Task 2. The forward pass compiled – here is the output when running the forward pass on the `wine-red.csv` data.

```
**Forward**


X.shape:          (595, 11)
Y.shape:          (595, 3)


*Layer 1*

W_L1.shape:       (11, 8)
b_L1.shape:       (8,)

XW1.shape:        (595, 8)
plus_b1.shape:    (595, 8)
Z_1.shape:        (595, 8)

*Layer 2*

W_L2.shape:       (8, 8)
b_L2.shape:       (8,)

W2Z1.shape:       (595, 8)
plus_b2.shape:    (595, 8)
Z_2.shape:        (595, 8)

*Layer 3*

W_L3.shape:       (8, 3)
b_L3.shape:       (3,)

W3Z2.shape:       (595, 3)
plus_b.shape:     (595, 3)
Y_hat.shape:      (595, 3)
Loss.shape:       (595, 3)
```

And when calling y_hat directly:

```
array([[0.00056447, 0.0005568 , 0.00055941], [0.00056447, 0.0005568 , 0.00055941], [0.00056447,
0.0005568 , 0.00055941], ..., [0.00056447, 0.0005568 , 0.00055941], [0.00056447, 0.0005568 ,
0.00055941], [0.00056447, 0.0005568 , 0.00055941]])
```

And loss:

```
array([[ 0. , 0. , -7.4886287 ], [ 0. , 0. , -7.48863197], [ 0. , 0. , -7.48863456], ..., [ 0. , 0. , -
7.48863404], [ 0. , 0. , -7.48863364], [ 0. , 0. , -7.48863467]])
```

There seems to be a problem – the softmax output, reflected in the y_hat array, do not sum to one for each observation. There are a lot of computations in this notebook, and there were many opportunities to make mistakes. This requires further investigation.

Moreover, the backward pass did not compile, due to incompatible matrix shapes when using the chain rule:

```
**Backward**


d_yhat.shape:        (595, 3)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-613-114c1f279e19> in <module>
----> 1 y_hat, loss = this_cg.backward()

---------------------------------------------------------------------------
1 frames
---------------------------------------------------------------------------
<ipython-input-605-8b89dd843ac7> in backward(self, upstream_g)
     41                 indicator = 1 if i==j else 0
     42                 local_g[i] = Z[i]*(indicator-Z[i])
---> 43         dM = np.dot(local_g,upstream_g)
     44         return dM
     45

<__array_function__ internals> in dot(*args, **kwargs)

ValueError: shapes (595,3) and (595,3) not aligned: 3 (dim 1) != 595 (dim 0)
```

Again, this could be due to errors in a number of places, and I didn't have time to figure it out. It is difficult to tell exactly where things went wrong with the forward and backward passes, since each method chains together a string of operations, with each output dependent on the previous.

Because of the issues with the backward pass, calling the train() method will produce errors similar to those above.

I am able to call the test() method, in the sense that it will compile, but the results cannot be trusted because of the issue I noticed with the forward pass. Regardless, here is the output when I call train() on the wine data:

```
**Forward**


X.shape:          (149, 11)
Y.shape:          (149, 3)


*Layer 1*

W_L1.shape:       (11, 8)
b_L1.shape:       (8,)

XW1.shape:        (149, 8)
plus_b1.shape:    (149, 8)
Z_1.shape:        (149, 8)

*Layer 2*

W_L2.shape:       (8, 8)
b_L2.shape:       (8,)

W2Z1.shape:       (149, 8)
plus_b2.shape:    (149, 8)
Z_2.shape:        (149, 8)

*Layer 3*

W_L3.shape:       (8, 3)
b_L3.shape:       (3,)

W3Z2.shape:       (149, 3)
```

```
plus_b.shape:    (149, 3)
Y_hat.shape:     (149, 3)
Loss.shape:      (149, 3)

y_hat_test:
 [[ 0.          0.         -6.09808715]
 [ 0.          0.         -6.09808827]
 [ 0.         -6.10031433  0.        ]
 [ 0.         -6.10031691  0.        ]
 [ 0.          0.         -6.09808193]
 [ 0.          0.         -6.09808629]
 [ 0.          0.         -6.09808678]
 [ 0.          0.         -6.09808784]
 [ 0.          0.         -6.09808704]
 [ 0.          0.         -6.09808161]
 [ 0.          0.         -6.09808637]
 [ 0.         -6.10031354  0.        ]
...
 [ 0.          0.         -6.09808849]
 [ 0.          0.         -6.09808709]
 [ 0.          0.         -6.09808561]
 [ 0.          0.         -6.09808489]
 [ 0.          0.         -6.09808381]
 [ 0.          0.         -6.09808735]
 [ 0.          0.         -6.09808538]
 [ 0.          0.         -6.09808564]
 [ 0.         -6.10031471  0.        ]
 [ 0.          0.         -6.09808554]
 [ 0.          0.         -6.09808514]]

test_loss:
 0.8859060402684564
```

I suspect that I need to brush up on linear algebra, or at least linear algebra in the context of neural networks.

5 References

[1] Chollet, F. (2017). Deep learning with python. Manning Publications.