CS 577: Final Project

Addernets - Reformulating CNNs without Multiplications

Jane Downer - A20452471
Quinlan Bock - A20492935

November, 15, 2022

Abstract

A discussion and implementation of AdderNet [1], an architecture that replicates the behavior of convolutional neural networks via matrix addition rather than multiplication.

Team Member Responsibilities

Code

- Jane: Adder Layer, Flatten Layer, Fully Connected Layer, Max Pool Layer, and Model Class.
- Quin: Batch Norm Layer, and Minibatch
- Both: Collaborative debugging/testing/plotting

Paper/Presentation

- Jane: Proposed Solution, Results and Discussion
- Quin: Problem Statement, Implementation Details, Presentation

Problem statement

Multiplications in neural networks are computationally expensive and slow the training process. This makes it difficult to train deep and complex networks, particularly with CNNs, and prevents improvements in efficiency. Multiplication of tensors in neural networks makes it impractical not to train on GPU or TPU; these devices are not as ubiquitous as other chip architectures, are power intensive, and expensive. AdderNet [1] thus relieves the dependence on such devices and opens up opportunities for applications in edge devices.

Proposed Solution

AdderNet proposes several modifications to the traditional convolutional neural network.

1. Abandoning convolution in favor of addition-based operations. AdderNet proposes using $l1$l-distance as a similarity measure between windows and filters.

$$l1(a, y) = \sum_{i=1}^{n} |x_i - y_i| \qquad \text{Eq. 1}$$

2. Update filters using full-precision gradients, which the paper's authors argue more frequently take the direction of steepest descent and are more suitable for networks with many parameters.

$$\frac{\partial Y(i,m,n,t)}{\partial F(i,j,k,t)} = X(m + i, n + j, k) - F(i, j, k, t) \qquad \text{Eq. 2}$$

3. Also use the full-precision gradient for network inputs, but clipped to range (-1,1) since it affects all prior layers via the chain rule.

$$\frac{\partial Y(i,m,n,t)}{\partial X(m+i,n+j,k)} = HT(F(i,j,k,t) - X(m+i,n+j,k)) \quad \text{Eq. 3}$$

Where

$$H(x) = \begin{matrix} x & if\ -1 < x < 1 \\ 1 & if\ x >= 1 \\ 0\ otherwise \end{matrix} \quad \text{Eq. 4}$$

4. The nature of the addition-based operations results in larger variance for layer outputs. AdderNet uses batch normalization to "promote more effective activations". Batch-normalized output can be calculated by:

$$y = \frac{\gamma x - \mu_B}{\sigma_B}\beta \quad \text{Eq. 5}$$

where B denotes a batch and γ and β are learnable parameters.

5. The larger variance mentioned above can yield smaller gradient norms. To counteract this, AdderNet introduces the concept of an additional, adaptive gradient rate for each network layer l, defined as:

$$\alpha_l = \frac{\eta\sqrt{k}}{\left\|\Delta L(F_l)\right\|_2} \quad \text{Eq. 6}$$

Where k is the number of elements in filter F_l, $\Delta L(F_l)$ is the change in loss with respect to filter F_l, and η is a hyper-parameter. The update rule for the filter is thus

$$\Delta F_l = \gamma \times \alpha_l \times \Delta L(F_l) \quad \text{Eq. 7}$$

Implementation Details

Our entire network implementation was done from scratch without the use of any deep learning libraries. NumPy was used for higher level numerical calculations and for speeding up computation that could be handled by Python. Similarly SciPy was used for additional functionality. The main functionality lies in model.py and layers.py where the model class and the layers classes were respectively defined. A data loader was defined in data.py and the rest of smaller functions fell under util.py.

Utils

We implemented a number of layer classes to form the backbone of our model.

- L1() - returns the L1 distance between two points by taking the difference between its two inputs and returning the absolute value of that difference using np.abs(). See Eq 1.
- hard_tanh() - The hard tangent function is implemented using np.clip() and specifying the upper and lower bounds to clip at as 1 and -1 respectively. See Eq 4.
- eps() - returns a small value (epsilon) from a uniform distribution with a lower bound of 1e-7 and an upper bound of 1e-6. This function is used repeatedly to allow use of natural logs, and division where zeros would make the operations undefined..
- cat_cross_entropy() - computes the categorical cross entropy between the true one hot encoded label and a prediction. It does so by using numpy to compute the log of prediction array with an added epsilon, then multiplying the result with the ground truth array, and summing over the results and dividing to take the average value over all the examples in the batch.
- cat_cross_entropy_prime() - computes the local gradient of the categorical cross entropy function with respect to its input. This is done by returning an array of the negative ratio between the growth truth and the prediction using list comprehension.

- get_mini_batches() - takes as its arguments, input data, target data, and the batch_size to split this data up into. The function operates through a loop that generates upper and lower indices used to split the input and target data and append them to a list of batches and is returned.

Data

A function load_cifar_data is written to aid in downloading, importing, and preprocessing CIFAR-10 data. It includes options to specify the folder the data is stored in, whether to download the data and store it to a folder, and whether to return the full dataset, or a smaller subset of the dataset. If the option is selected to download the dataset the wget package is used to the download the .tar file, which is then unzipped and moved into a folder using the tarfile and os packages. The data is stored in multiple pickle files which are loaded and appended using the pickle package. The final option to return a smaller dataset works by slicing the data array and returning them, otherwise the full array is returned.

Layers

We implemented a number of layer classes to form the backbone of our model.

- Layer() – the parent class that defines the structure of more specific layers. All layers include forward and backward methods and store the inputs to the layer and the outputs of the layer. The forward takes as its arguments the input to the layer, and a boolean value: init_weights which is specified to only be true on the first forward pass through the network and tells the forward function to initialize weights for that layer.
- adder_layer() – takes number of output channels, kernel size, stride, padding, and eta (see Eq. 6) as hyperparameters. Includes the following methods:
    - __init__() - initializes variable passed in the object declaration, additionally it initializes the filters and bias variables using np.ones() and np.zeros() respectively.
    - get_adaptive_learning_rate() - computes the adaptive learning rate according to Eq. 6
    - forward() - iterates over image tensor in 'sliding-window' fashion, computing the dot product between the filter and the input window and entering the result into the output tensor.
    - backward() - takes an upstream gradient and learning rate as input. Slides window across image tensor, computing the gradients with respect to teh image, weights, and biases. Computes the gradient with respect to the weights and images according to Eq. 2 and Eq. 3, respectively. After iterating over the image tensor, calls self .get_adaptive_learning_rate(). Updates filters and biases with the below rule, and returns the gradient with respect to the input.

      Update rules:

      ```
      self.filters -= learning_rate*adaptive_lr*dfilters
      self.bias    -= learning_rate*dbias
      ```
- FullyConnected() – standard 'dense' layer. Initialized with number of output channels.
    - forward() - Performs a single matrix multiplication (image tensor times weight tensor), adds bias, and returns result.
    - backward() - takes upstream gradient and learning rate as input. Calculates gradients with respect to X, W, and bias with the below formulas:

      ```
      dX    = np.dot(upstream_g, self.weights.T)
      dW    = np.dot(self.input.T, upstream_g)
      ```

```
dbias = np.mean(upstream_g)
```

And updates weights and bias with the following rules:

```
self.weights -= learning_rate*dW
self.bias    -= learning_rate*dbias
```

- Returns gradient with respect to X.

- Flatten() - while technically a layer, mainly serves to reshape X into an array-like output. Gradient and and individual elements do not change with forward or backward pass.
- batch_norm_layer() - normalizes a four-dimensional 'batch' of images during training, with learnable scale and shift parameters.
  - forward() - computes mean and standard deviation of inputs and normalizes the data. Scales by 'gamma' and shifts by 'beta', which are instantiated to ones and zeros so as to not have any effect unless the model updates them in the backward pass.
  - backward() - takes upstream gradient and learning rate as input. Computes the gradients with respect to gamma, beta and inputs via the following equation:

```
for i in range(m):
    for j in range(m):
        dX[i] += (upstream_g[i] - upstream_g[j]*(1 + (X[i]-X[j])*(X[j]-mean)/std))
dX *= self.gamma/((m**2)*std)
```

    And updates the gradients:

```
self.gamma = self.gamma - learning_rate*dGamma
self.beta  = self.beta  - learning_rate*dBeta
```

    Returns gradients with respect to the input.
- MaxPool() – takes max pool filter size and stride as hyperparamater inputs.
  - forward() - performs a sliding window search over image tensor and computes the maximum at each location. Populates a smaller tensor with these values and outputs the result. Keeps track of locations of maxima for use in backward pass.
  - backward() - outputs a gradient the shape of the original tensor which is 1 at the locations of the maxima, and 0 otherwise.
- Activation() -takes 'relu' and 'softmax' as options.
  - forward() - performs non-linear transformation on output of previous layer.
    - 'Relu':
      - Changes any negative elements to zero and outputs result.
    - 'Softmax':
      - Takes the output of a fully-connected layer, which has 10 output channels (one per class). Converts the 10 elements in each example into 'probabilities' via the following formula and outputs the result.

```
exp_ = np.exp(x - np.max(x))
        return exp_ / (np.sum(exp_) + eps())
```

  - backward() - takes upstream gradient and learning rate as input, computes gradients with respect to input, and returns result
    - 'Relu':
      - Computes gradient by converting the positive values of the upstream gradient to 1, and the negatives to 0.
    - 'Softmax'
      - Instantiates a matrix of zeros in the shape of softmax(input). Sets each element `(i,j)` of the matrix equal to:

```
J[i][j] = s[i][j]*(indicator_ij - s[i][j])
```

- Where `indicator_ij` is equal to one if i==j, else 0.

Model

The model class allows for building of sequential models. It takes as it's input only the name of the name of the loss function to use with the model.

- add() - appends layer input to the self.layers.
- predict() - returns a prediction for the input passed to the function. It works by calling looping the layers in self.layer and calling forward on each of the layers, successively pushing the input through the network and finally it returns the result of the network.

```
Z = input_data
for layer in self.layers:
        Z = layer.forward(Z,init_weight=False)
```

- fit() - this function does the training on the model for the specified number of epochs, with minibatch. The function works through two main loops, the first of which iterates over the number of epochs it is training for, before the second loop starts get_mini_batches is called and the result stored so that the second loop can iterate over the batches in the dataset. Within the second loop the batch first pushed through the network in the same way a single example in predict() was. Then the loss on this batch is calculated and the backwards pass is started be executing:

```
error = -y/(np.argmax(y_pred,axis=1) + util.eps())
for layer in (self.layers)[::-1]:
        Error = layer.backward(error,learning_rate)
```

Once the forwards and backwards pass is completed for every batch then the accuracy is computed and the loss and the accuracy are appended to the history variable and the validation loss and accuracy is computed in the same way and printed out. Once all the epochs are complete the function returns the history.

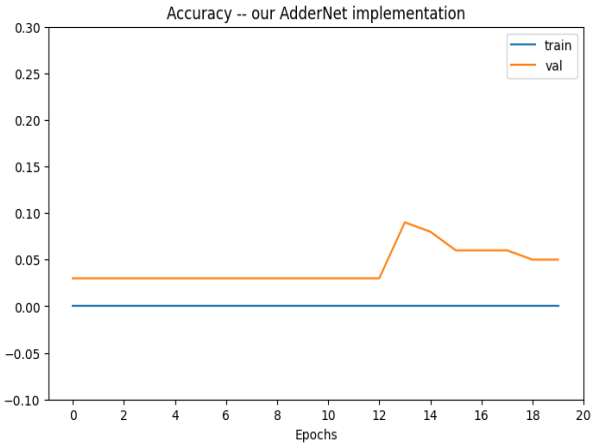
Results and Discussion

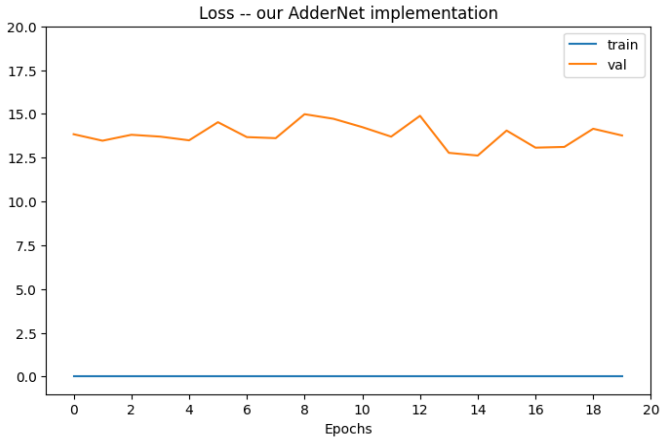
Part I. Our Implementation

The most straightforward way to assess the performance of our implementation is to train the model on image data and plot the resulting accuracy and loss. This proved to be a difficult task, first, because of the computational expenditure required by the sliding window addition – in our implementation, this involved a nested loop that traversed every dimension of the input. As we mention below, the problem with speed was not unique to our implementation, and we faced these issues with the original AdderNet as well. As a result, we often trained with a small dataset to monitor performance progress. Below is the result of one such training loop.

```
Training set: 100 images
Validation set: 100 images
Epochs: 20
Batch size: 16
```

```
Learning rate: 1e-05
```






```
Test loss: 12.814556466007623
```

It unfortunately looks like not much learning is happening. We explored a number of reasons for why this could be the case. A few mistakes that we found, and corrected:

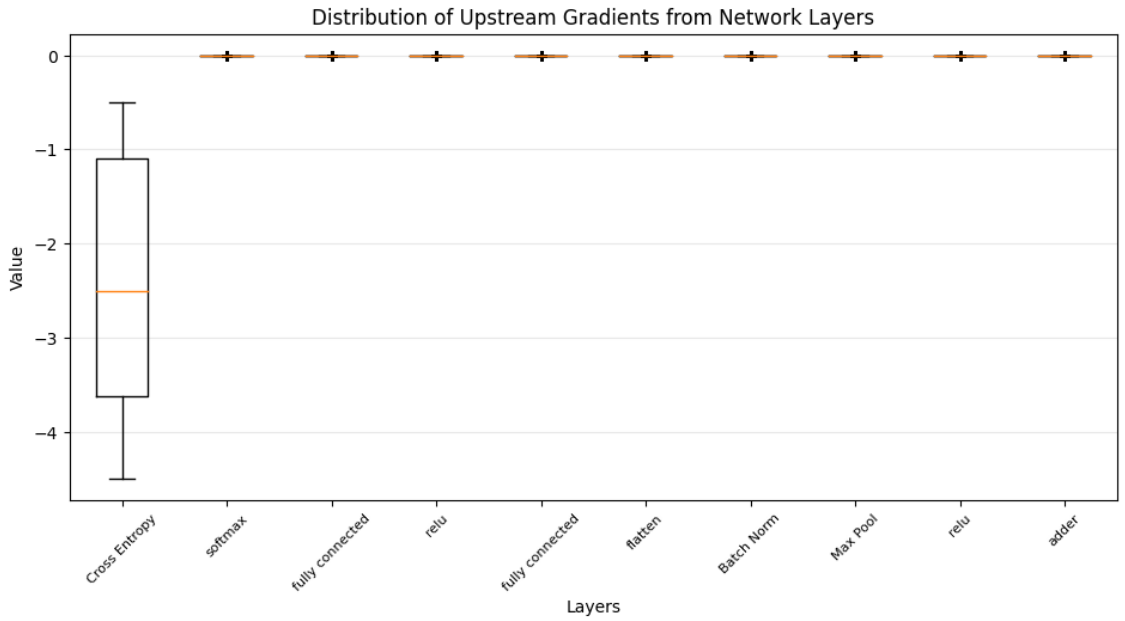
- The equations for gradients with respect to weights and inputs (Eq. 2 and Eq. 3) were reversed – this was corrected
- Weights were re-initialized within each forward pass – corrected
- np.log() and division operations were being called on 0 – corrected by adding small value, eps() (discussed above)
- Softmax function was unstable – corrected by includign the np.max() term below:

```
np.exp(x - np.max(x)) / (np.sum(exp_) + eps())
```

- We also explored a number of combinations fo hyperparameters, with no change to our results.

There are two currently-apparent issues.

1. Training loss is consistently reported to be zero. Within the training loop, we average the loss as we iterate over images in a batch. It is possible that we have overlooked an instance where we divided unnecessarily, thus reducing the loss value to a small, near-zero value. This is likely the case, because we are able to call loss() on the output of a forward pass, and obtain non-zero values.
2. If learning is not happening, then there is very likely an issue with the gradients. Below is a visualizaiton of the gradient values computed by each layer in the backward pass:



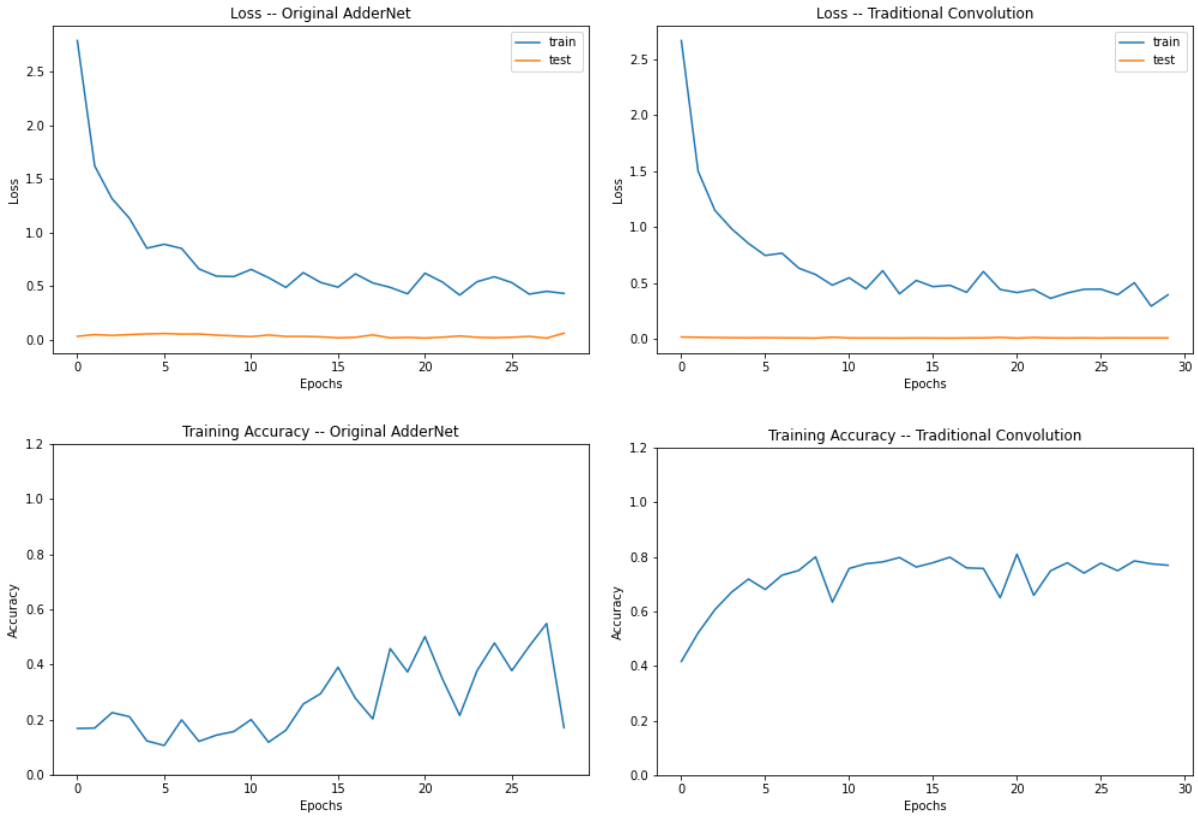
This shows that after the softmax.backward() pass, the gradients with respect to input immediately drop to zero. This was not unique to a particular epoch. There is clearly an issue with the backward method within the softmax activation layer. Here is a closer look at this method:

```
def backward(self, upstream_g, learning_rate):

    local_g = None
    if self.activation_name == 'relu':
        local_g = np.where(self.input>=0,1,0)

    elif self.activation_name == 'softmax':
        s = self.output
        J = np.zeros_like(s)
        for i in range(len(s)):
            for j in range(len(s[i])):
                indicator_ij = 1 if i==j else 0
                J[i][j] = s[i][j]*(indicator_ij - s[i][j])
        local_g = J

    dX = np.zeros_like(self.input)
    for i, (l,u) in enumerate(zip(local_g, upstream_g)):
        dX[i]=learning_rate*l*u

    return dX
```

The output of this method involves the multiplication of local and upstream gradients. It is possible that these values are already small, and that their multiplication reduces them to near zero.

Part II. Original Addernet

A purported benefit of AdderNet is its alleged speed over regular convolution. The Addernet repository includes a model with its own “adder2d” layers, which replace convolution. By substituting these layers for tf.keras.Conv2D layers, we can easily compare performance against a traditional, convolution-based model. Below are the results:



After 30 epochs:

Original AdderNet:

```
Train Loss: 0.514889    Accuracy: 0.378600      Test Loss: 0.041237 time
spent: 64 minutes and 2 seconds
```

Replacing ‘adder’ layers with traditional convolution:

```
Train Loss: 0.392941    Accuracy: 0.768900      Test Loss: 0.007318
time spent: 3 minutes and 55 seconds
```

We were surprised to find that, despite the reported benefits of AdderNet, we did not find improvement in speed or accuracy. In fact, a traditional convolution-based network trained over 10 times as fast, and with far superior accuracy. We expected accuracy to suffer slightly, which is acknowledged in the paper, but had the assumption that it would be made up for with time and efficiency. Unfortunately this was not the case.

Future Improvements

Given more time, we would like to sort out the issues with the softmax backward() function so learning can occur. We would also like to test against our own implementation of a convolution-based network, by substituting the adder layers in our network, as we did with the network from the original AdderNet repository. We would also like to test the runtime of various functions within the original AdderNet code to find the source of the slowdown.

References

[1] Hanting Chen, Yunhe Wang, Chunjing Xu, Boxin Shi, Chao Xu, Qi Tian, and Chang Xu. Addernet: Do we really need multiplications in deep learning? CVPR, 2020. https://openaccess.thecvf.com/content_CVPR_2020/papers/Chen_AdderNet_Do_We_Really_Need_Multiplications_in_Deep_Learning_CVPR_2020_paper.pdf

[2] Alex Krizhevsky, Vinod Nair, and Georey Hinton. Cifar-10 (Canadian Institute for Advanced Research). https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz