AdderNets

Reformulating CNNs without Multiplications

Jane Downer, Quinlan Bock

Need for Lightweight Networks

- Low power, smaller devices without GPU or TPU
  - Mobile devices
  - IOT
- Settings where low latency is essential
- Addernet aims to reduce the computational complexity of CNNs
- Large matrix operations -> addition operations

Mathematics - adder.forward()

"Sliding window" operation to measure similarity – like convolution

- The difference: addition-based

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

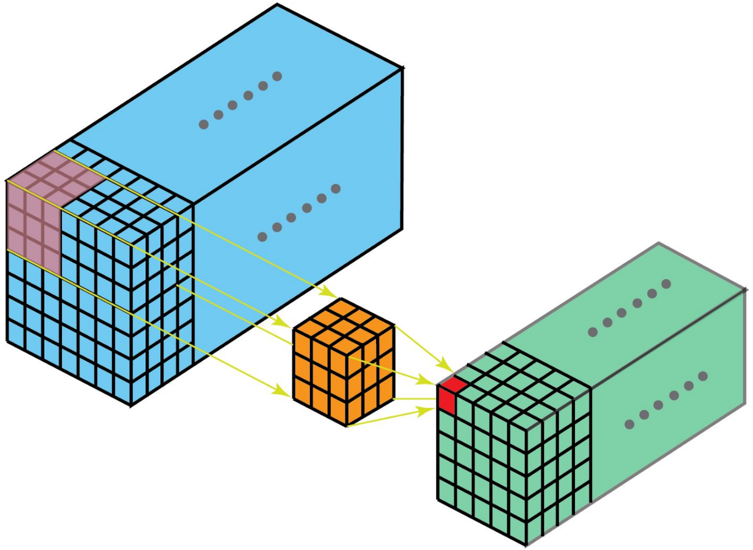


Image source: Towards Data Science [1]

Mathematics - adder.backward()

Full-precision gradients

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

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

Mathematics - Adaptive Learning Rate

Gradient Update Rule:

$$\Delta F_l = \gamma \times \alpha_l \times \Delta L(F_l)$$

Adaptive Learning Rate:

$$\alpha_l = \frac{\eta\sqrt{k}}{\left\|\Delta L(F_l)\right\|_2}$$

Implementation - Layers

- All layers implemented from scratch
- Subclass Layer class, must define forward and backward functions
- Adder, Dense, Flatten, Batch Norm, MaxPool, Activation (relu, softmax), Conv

```
this_model = Model(loss_name='cat_cross_entropy')

this_model.add(layers.adder_layer(output_channels=8,kernel_size=3,stride=1,padding=1,adaptive_eta=0.1))
this_model.add(layers.Activation('relu'))
this_model.add(layers.MaxPool(pool_size=2))
this_model.add(layers.batch_norm_layer())
this_model.add(layers.Flatten())
this_model.add(layers.FullyConnected(output_channels=64))
this_model.add(layers.Activation('relu'))
this_model.add(layers.FullyConnected(output_channels=10))
this_model.add(layers.Activation('softmax'))
```

Implementation - Layers

Adder_layer.backward()

```
for i in range(n_tensors):
          x = X_padded[i]
          dx = dX_padded[i]

      #x, dx = X_padded, dX_padded
          for h in range(H_up):                    # traverse height
              for w in range(W_up):                # traverse width
                  for c in range(c_up):            # traverse filters

                      v0,v1 = h,h+H_k
                      h0,h1 = w,w+W_k

                      x_window = x[v0:v1, h0:h1, :]
                      f_window = filters[c,:,:,:]

                      dx_local = hard_tanh(f_window-x_window)
                      df_local = x_window-f_window

                      g = upstream_g[i, h, w, c]

                      dx[v0:v1, v0:v1, :] += dx_local * g
                      dfilters[c,:,:,:]   += df_local * g
                      dbias[c,:,:,:]      += g
```

Adder_layer.forward():

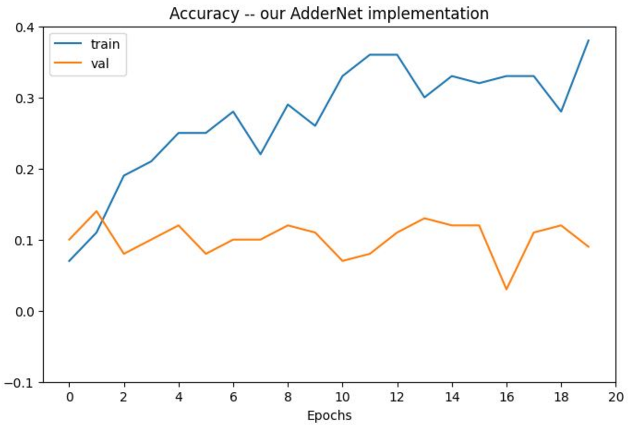
```
Z = np.zeros([n_tensors, H_new, W_new, c_out])

for i in range(n_tensors):            # traverse batch
     this_img = X_padded[i,:,:,:]     # select ith image in batch
     for f in range(n_filters):       # traverse filters
          this_filter = filters[f,:,:,:]
          this_bias = bias[f,:,:,:]
          for h in range(H_new-H_k):   # traverse height
               for w in range(W_new):   # traverse width
                    v0,v1 = h*stride, h*stride + H_k
                    h0,h1 = w*stride, w*stride + W_k
                    this_window = this_img[v0:v1,h0:h1,:]
                    Z[i, h, w, f] = np.abs(this_window-this_filter).sum()
```

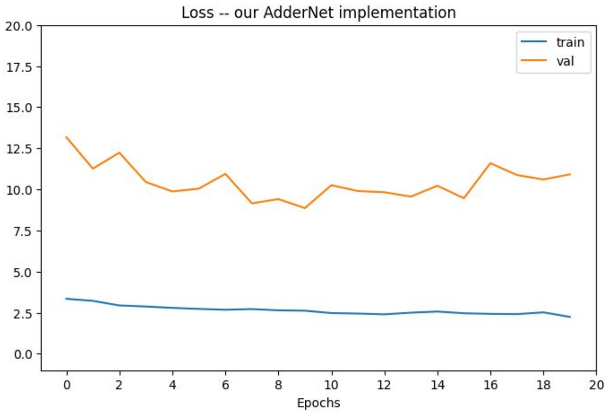
Implementation - Model

Model.fit()

```
for e in range(epochs):

    loss_,acc,val_loss,val_acc=0,0,0,0
    mini_batches = util.get_mini_batches(x_train,y_train, batch_size)

    for i, mini_batch in enumerate(mini_batches):

        x_batch = mini_batch[0]
        y_batch = mini_batch[1]

        # forward
        Z = x_batch

        for layer in self.layers:
            init_weights=True if e==0 else False
            Z = layer.forward(Z,init_weights=init_weights)

        # backward
        y_real, y_pred = y_batch, Z
        error = -np.argmax(y_real,axis=1)/(np.argmax(y_pred,axis=1) + util.eps())

        for layer in (self.layers)[::-1]:

            error = layer.backward(error, learning_rate)

        loss_ /= x_train.shape[0]
        acc  /= x_train.shape[0]

        if x_val is not None:

            Z_val = x_val

            for layer in self.layers:
                Z_val = layer.forward(Z_val,init_weights=False)

            Y_real_val, y_pred_val = y_val, Z_val
            val_loss = self.loss_fwd(y_real_val, y_pred_val
            val_acc = sum(np.where(np.argmax(y_real_val,axis=1)==np.argmax(y_pred_val,axis=1),1,0)
            val_acc  /= x_val.shape[0]
```

Results - Our Implementation

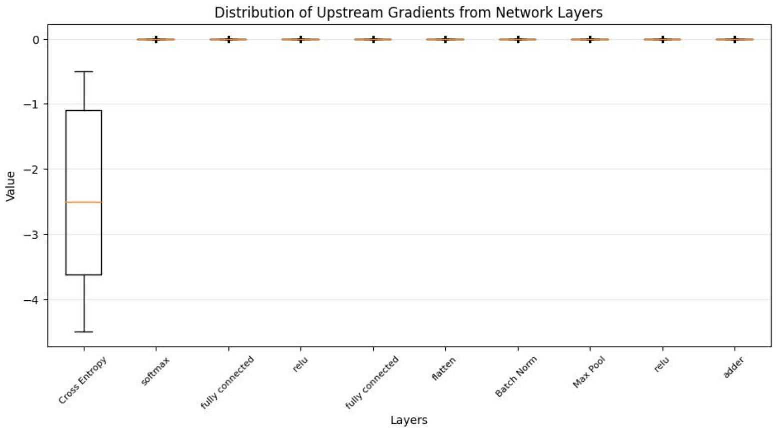


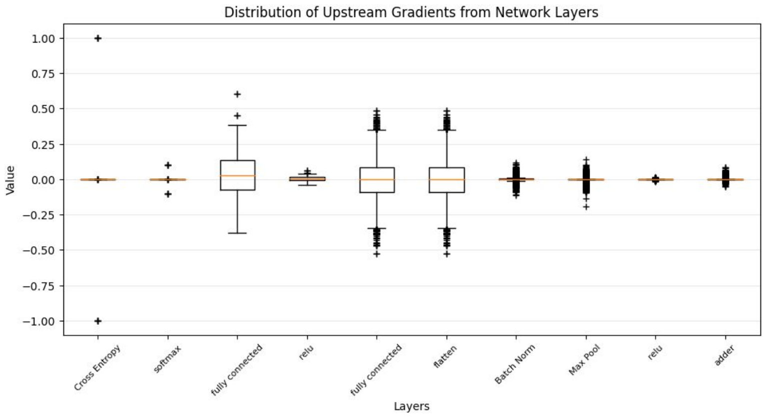


- Epochs: 20 Batch Size: 16 Learning Rate: 1e-3
- Were able to begin to overfit the data

Results - Our Implementation

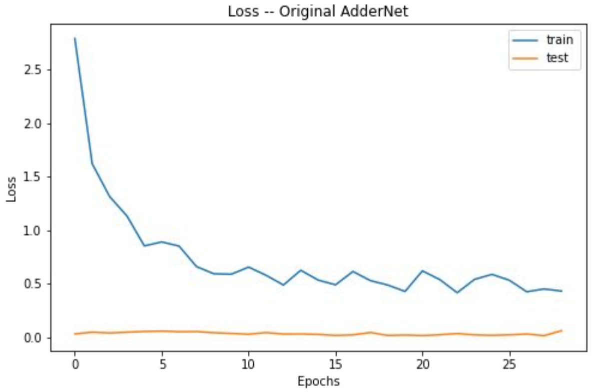
- Upstream gradients drop to 0 after softmax.backward()
- Plotting gradient distributions helped debug the backwards pass

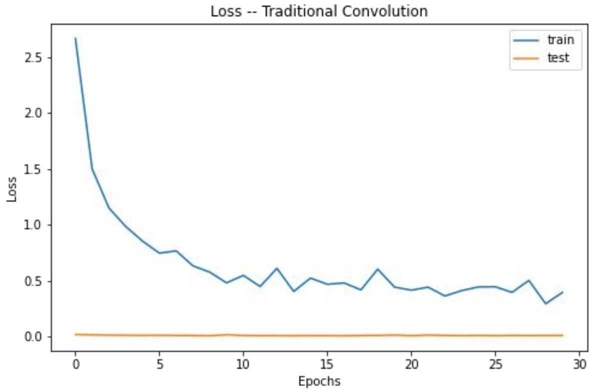


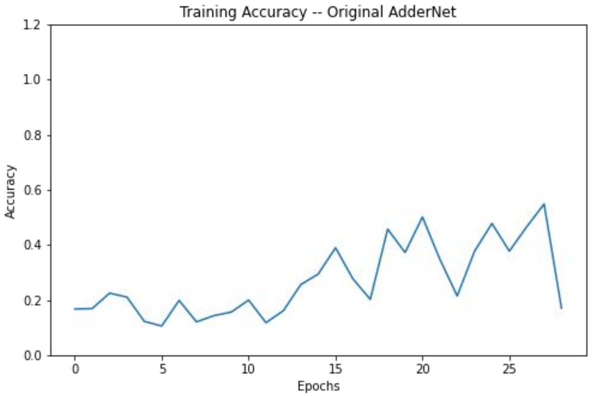


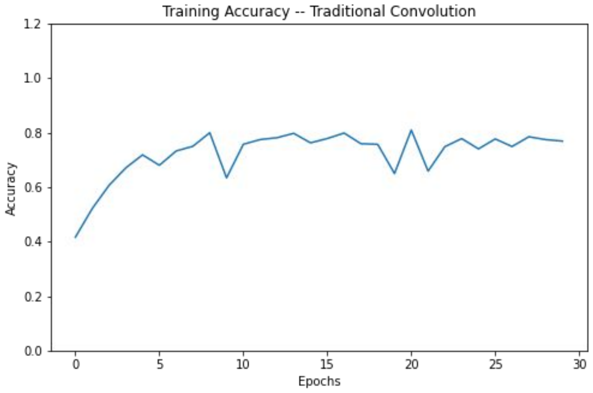
Results - Author's Implementation

- Original network and identical network with regular keras Conv2D layers in place of the adder layers
- The original network saw less learning and a lower accuracy in the 30 epochs that both networks where trained
- The original networks finished 30 epochs in ~3 minutes as opposed to the addernet that took ~60 minutes









References

[1] https://towardsdatascience.com/a-comprehensive-introduction-to-different-types-of-convolutions-in-deep-learning-669281e58215

[2]

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

[3]

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