Assignment 1

Martynas Vaznonis (2701013)

November 2023

1 Answers

question 1 The partial derivative for loss is

$$\frac{\partial l}{\partial y_i} = \frac{\partial}{\partial y_i}(-\log(y_i)) \begin{cases} -\frac{1}{y_i} & \text{if } i = c \\ 0 & \text{otherwise} \end{cases}$$

The partial derivative for softmax when $i = j$ is

$$\frac{\partial y_i}{\partial o_i} = \frac{\partial}{\partial o_i}\left(\frac{\exp(o_i)}{\sum_j \exp(o_j)}\right) = \frac{\exp(o_i)\sum_j \exp(o_j) - \exp(o_i)\exp(o_i)}{(\sum_j \exp(o_j))^2} = \frac{\exp(o_i)\sum_{j\neq i} \exp(o_j)}{(\sum_j \exp(o_j))^2}$$

The partial derivative for softmax when $i \neq j$ is

$$\frac{\partial y_i}{\partial o_j} = \frac{\partial}{\partial o_i}\left(\frac{\exp(o_i)}{\sum_j \exp(o_j)}\right) = \frac{-\exp(o_i)\exp(o_j)}{(\sum_j \exp(o_j))^2}$$

question 2 The derivative for loss with respect to pre-activation output is

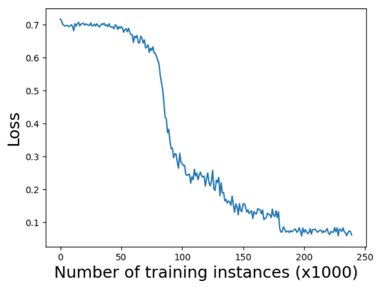
$$\frac{\partial l}{\partial o_i} = \frac{\partial l}{\partial y_j}\frac{\partial y_j}{\partial o_i} = \begin{cases} 0 & \text{if } j \neq c \\ -\frac{\exp(o_i)\sum_{j\neq i}\exp(o_j)}{y_j(\sum_j \exp(o_j))^2} & \text{if } i = j \\ \frac{\exp(o_i)\exp(o_j)}{y_j(\sum_j \exp(o_j))^2} & \text{if } i \neq j \end{cases}$$

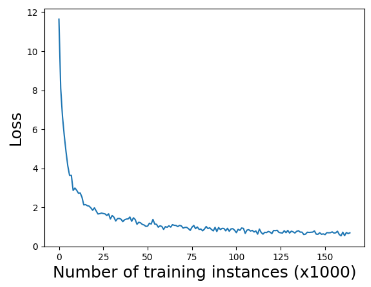
It is not strictly necessary to work out the global derivative because once the loss is calculated, the local derivatives can be evaluated. For example, if y_j is known, then $\frac{\partial l}{\partial y_j}$ takes on some concrete value x which can then be used in $\frac{\partial l}{\partial o_i} = \frac{\partial l}{\partial y_j}\frac{\partial y_j}{\partial o_i} = x\frac{\partial y_j}{\partial o_i}$.

question 3 The code for the forward and backward passes has been implemented inside of a class and can be seen below. Alongside the two methods, I have pasted the helper functions.

```python
def forward(self, x, y):
    self.data, self.target = x, y
    self.h = self.next_layer(x, self.W, self.b, func=sigmoid)
    self.y = softmax(self.next_layer(self.h, self.V, self.c))
    self.losses.append(-log(self.y[y]))


def backward(self):
    # Find derivatives
    dloss = -1/self.y[self.target]
    do = [dloss * self.y[i] * ((i == self.target) -\
          self.y[self.target]) for i in range(len(self.y))]
    dV = [[o * h for o in do] for h in self.h]

    dh = [sum([o * v for o, v in zip(do, V)]) for V in self.V]
    dh = [d * h * (1 - h) for d, h in zip(dh, self.h)]
    dW = [[i * h for h in dh] for i in self.data]

    # Update weights
    for j in range(len(self.V[0])):
        for i in range(len(self.V)):
            self.V[i][j] -= self.lr * dV[i][j]
        self.c[j] -= self.lr * do[j]

    for j in range(len(self.W[0])):
        for i in range(len(self.W)):
            self.W[i][j] -= self.lr * dW[i][j]
        self.b[j] -= self.lr * dh[j]

def sigmoid(x):
    return 1/(1+e**-x)

def softmax(x):
    x = [e**o for o in x]
    E = sum(x)
    return [o/E for o in x]

def next_layer(I, W, B, func=lambda x:x):
    o = [0] * len(B)
    for inp, ws in zip(I, W):
        for i, w in enumerate(ws):
            o[i] += w * inp
    for i in range(len(o)):
        o[i] = func(o[i] + B[i])
    return o
```

The derivative values wrt **W**, b, **V**, and c are the same as the ones expected in the assignment description.


(a) The loss curve for the synthesized data


(b) The loss curve for the MNIST dataset

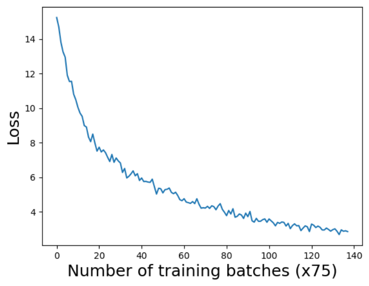
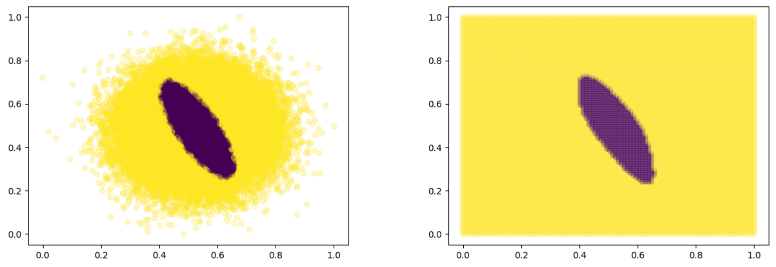

(c) The loss curve for the MNIST data with minibatch gradients and batch size set to 16

Figure 1

question 4 First the data is loaded in, concatenated, and normalized as seen below. Figure 2a shows the shape of the data.

```
(xtrain, ytrain), (xval, yval), num_cls = load_synth()
t = np.concatenate([xtrain, xval])
min, max = t.min(axis=0), t.max(axis=0)
xtrain -= min
xtrain /= max - min
xval -= min
xval /= max - min
```

Then, the training loop is simply:



(a) The synthesized data distribution (b) The learned classification boundary

Figure 2

```
fcn = ScalarFCN()
EPOCHS = 1
for i in range(EPOCHS):
    print(f"Epoch {i + 1}/{EPOCHS}")
    for x, y in tqdm(zip(xtrain, ytrain)):
        fcn.forward(x, y)
        fcn.backward()
```

The result can be seen in figure 1a which shows that the loss quickly drops and then remains noisy because it is calculated per instance and the network is tiny. Furthermore, figure 2b shows the classification boundary discovered by the network.

question 5 The code for the neural network has been implemented as a class:

```
class VectorFCN:
    def __init__(self,
                 input_size=784,
                 hidden_size=300,
                 output_size=10,
                 lr=1e-3):
        self.lr = lr
        self.W = np.random.randn(hidden_size, input_size)
        self.b = np.zeros(hidden_size)
        self.V = np.random.randn(output_size, hidden_size)
        self.c = np.zeros(output_size)
        self.losses = list()

    def classify(self, x):
        return np.argmax(softmax(self.V @
                         sigmoid(self.W @ x + self.b) + self.c))
    def forward(self, inp, target):
        self.data, self.target = inp, target
        self.h = sigmoid(self.W @ inp + self.b)
        self.y = softmax(self.V @ self.h + self.c)
        self.losses.append(-np.log(self.y[target]))

    def backward(self):
        dloss = -1/self.y[self.target]
        do = dloss * self.y * ((np.arange(10) == self.target) -\
            self.y[self.target])
        self.c -= self.lr * do

        dV = do.reshape(-1, 1) @ self.h.reshape(1, -1)
        self.V -= self.lr * dV

        dh = do @ self.V * self.h * (1 - self.h)
        self.b -= self.lr * dh

        dW = dh.reshape(-1, 1) @ self.data.reshape(1, -1)
        self.W -= self.lr * dW
```

The helper functions:

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def softmax(x):
    x = np.exp(x)
    return x/sum(x)
```

Furthermore, the data values were normalized to the range $[0, 1]$ and the training loop is identical to the one outlined previously. Figure 1b shows the loss curve on the MNIST data with a validation accuracy of $\sim 88\%$.

question 6 Some minor adjustments are made to the forward and backward methods:

```
def forward(self, inp, target):
    self.data, self.target =\
        inp.reshape(-1, self.input_size, 1), target
    self.h = sigmoid(self.W @ self.data + self.b)
    self.y = softmax(self.V @ self.h + self.c)
    self.losses.append(-np.log(
        self.y[range(len(self.y)), target]).mean())
```

```
def backward(self):
    dloss = -1/self.y[range(len(self.y)), self.target]
    do = np.expand_dims(dloss, 1) * self.y * (
        (np.tile(range(10), (len(self.y), 1)) ==
         self.target.reshape(-1, 1)) -
        self.y[range(len(self.y)), self.target]
    ).reshape(-1, 10, 1)
    self.c -= self.lr * do.mean(axis=0)

    dV = do @ self.h.swapaxes(1, -1)
    self.V -= self.lr * dV.mean(axis=0)

    dh = self.V.T @ do * self.h * (1 - self.h)
    self.b -= self.lr * dh.mean(axis=0)

    dW = dh @ self.data.swapaxes(1, -1)
    self.W -= self.lr * dW.mean(axis=0)
```

As well as that, some other parts of the code needed to be corrected:

```
def softmax(x):
    x = np.exp(x)
    return x/np.expand_dims(x.sum(axis=1), -1)

self.b = np.zeros((hidden_size, 1))
self.c = np.zeros((output_size, 1))

def classify(self, x):
    return softmax(self.V @ sigmoid(
        self.W @ x.reshape(-1, self.input_size, 1) +
        self.b) + self.c
        ).reshape(-1, self.num_classes).argmax(axis=1)
```

Figure 1c shows the learning curve for the MNIST data with mini batches. The gradients are averaged rather than summed.

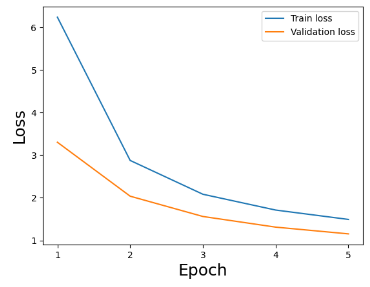
question 7 The batch size was set to 8 for every run in this section. The default learning rate was 0.001.

1. Figure 3a shows the average validation losses over epochs. This difference is important because it can show if the model is overfitting. The training loss may keep on dropping but if the validation loss begins increasing, it is a good indicator of overfitting.

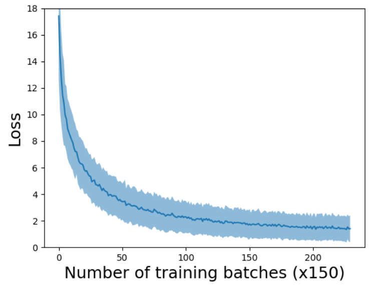
2. Figure 3b shows the learning curve averaged over 3 runs. It is useful in showing the stability of the model, i.e., that such an architecture can consistently achieve good results on similar problems.

3. Figure 3c shows the learning curves for the different learning rates. The larger the learning rate, the faster the model will be able to converge but it may overshoot optimal solutions by taking overly large steps. The best performance achieved was with learning rate set to 0.03. The smaller learning rates converge slower and thus have a weaker performance.

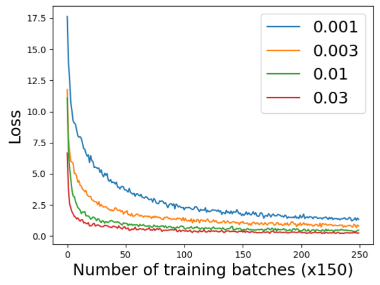
4. Figure 3d shows the learning curve when training over entire MNIST dataset averaged over 3 runs. The learning rate was set to 0.03 and the final accuracy achieved was $\sim 92\%$.



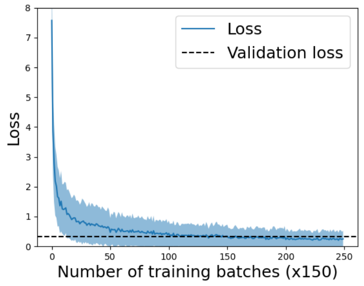
(a) Training versus validation loss over 5 epochs



(b) Learning curve averaged over 3 runs



(c) Learning curves from runs with varied learning rates



(d) The learning curve over the entire MNIST training data with 5 epochs averaged over 3 runs

Figure 3

A Appendix

- The gradients were tested at every step to ensure that the operations were correct. That means that once I calculated, for example, $\mathtt{h}^{\nabla}$, I tested that $\mathtt{softmax}(\mathbf{V}(\mathtt{h} - \mathtt{h}^{\nabla}) + \mathtt{c})$ does move in the direction of reducing loss.
- All losses in each learning curve have been grouped together by some number (e.g., 1000 or 75) and averaged to make the plots less noisy.
- The model for synthesized data had its learning rate reduced for the final epoch which lead to superior results.