Assignment 2

Martynas Vaznonis (2701013)

November 2023

1 Answers

question 1 All partial derivatives $\frac{\partial x_{ij}}{\partial y_{kl}}$ and $\frac{\partial y_{ij}}{\partial x_{kl}}$ only have nonzero values where $i = k \wedge j = l$, therefore

$$\frac{\partial \text{loss}}{\partial f(\mathbf{X},\mathbf{Y})}\frac{\partial f(\mathbf{X},\mathbf{Y})}{\partial \mathbf{X}} = \begin{bmatrix} \frac{\partial \text{loss}}{\partial \frac{x_{11}}{y_{11}}}\frac{\partial}{\partial x_{11}}(\frac{x_{11}}{y_{11}}) & \cdots & \frac{\partial \text{loss}}{\partial \frac{x_{1n}}{y_{1n}}}\frac{\partial}{\partial x_{1n}}(\frac{x_{1n}}{y_{1n}}) \\ \vdots & \ddots & \vdots \\ \frac{\partial \text{loss}}{\partial \frac{x_{m1}}{y_{m1}}}\frac{\partial}{\partial x_{m1}}(\frac{x_{m1}}{y_{m1}}) & \cdots & \frac{\partial \text{loss}}{\partial \frac{x_{mn}}{y_{mn}}}\frac{\partial}{\partial x_{mn}}(\frac{x_{mn}}{y_{mn}}) \end{bmatrix} =$$

$$\frac{\partial \text{loss}}{\partial f(\mathbf{X},\mathbf{Y})} \otimes \begin{bmatrix} \frac{1}{y_{11}} & \cdots & \frac{1}{y_{1n}} \\ \vdots & \ddots & \vdots \\ \frac{1}{y_{m1}} & \cdots & \frac{1}{y_{mn}} \end{bmatrix} = \frac{\partial \text{loss}}{\partial f(\mathbf{X},\mathbf{Y})} \otimes \frac{1}{\mathbf{Y}}$$

$$\frac{\partial \text{loss}}{\partial f(\mathbf{X},\mathbf{Y})}\frac{\partial f(\mathbf{X},\mathbf{Y})}{\partial \mathbf{Y}} = \begin{bmatrix} \frac{\partial \text{loss}}{\partial \frac{x_{11}}{y_{11}}}\frac{\partial}{\partial y_{11}}(\frac{x_{11}}{y_{11}}) & \cdots & \frac{\partial \text{loss}}{\partial \frac{x_{1n}}{y_{1n}}}\frac{\partial}{\partial y_{1n}}(\frac{x_{1n}}{y_{1n}}) \\ \vdots & \ddots & \vdots \\ \frac{\partial \text{loss}}{\partial \frac{x_{m1}}{y_{m1}}}\frac{\partial}{\partial y_{m1}}(\frac{x_{m1}}{y_{m1}}) & \cdots & \frac{\partial \text{loss}}{\partial \frac{x_{mn}}{y_{mn}}}\frac{\partial}{\partial y_{mn}}(\frac{x_{mn}}{y_{mn}}) \end{bmatrix} =$$

$$\frac{\partial \text{loss}}{\partial f(\mathbf{X},\mathbf{Y})} \otimes \begin{bmatrix} -\frac{x_{11}}{y_{11}^2} & \cdots & -\frac{x_{1n}}{y_{1n}^2} \\ \vdots & \ddots & \vdots \\ -\frac{x_{m1}}{y_{m1}^2} & \cdots & -\frac{x_{mn}}{y_{mn}^2} \end{bmatrix} = \frac{\partial \text{loss}}{\partial f(\mathbf{X},\mathbf{Y})} \otimes -\frac{\mathbf{X}}{\mathbf{Y}^2}$$

question 2 Because f is applied element-wise, for each y_i in $F(X)$ only x_i is important to find the derivative. Therefore, the derivative for y_i is $\frac{y_i}{x_i}$ and so

$$\frac{\partial \text{loss}}{\partial F(X)}\frac{\partial F(X)}{\partial X} = \begin{bmatrix} \frac{\partial \text{loss}}{\partial f(x_1)}\frac{\partial f(x_1)}{\partial x_1} \\ \vdots \\ \frac{\partial \text{loss}}{\partial f(x_n)}\frac{\partial f(x_n)}{\partial x_n} \end{bmatrix} = \frac{\partial \text{loss}}{\partial F(X)} \otimes \begin{bmatrix} f'(x_1) \\ \vdots \\ f'(x_n) \end{bmatrix}$$

question 3 Matrix multiplication $\mathbf{X}\mathbf{W}$ computes the outputs where $\mathbf{W}$ is a $f \times m$ matrix. Each row of $\mathbf{X}$ is multiplied with every column of $\mathbf{W}$, thus $o_{ij} = w_{1j}x_{i1} + w_{2j}x_{i2} + \ldots + w_{fj}x_{if}$. The scalar derivative with respect to w_{kj} then is $\frac{\partial o_{ij}}{\partial w_{kj}} = x_{ik}$. Hence, $\frac{\partial \text{loss}}{\partial w_{kj}} = \sum_i \frac{\partial \text{loss}}{\partial o_{ij}} x_{ik}$. Thus, the backward for $\mathbf{W}$ is

$$\frac{\partial \text{loss}}{\partial \mathbf{X}\mathbf{W}} \frac{\partial \mathbf{X}\mathbf{W}}{\partial \mathbf{W}} = \begin{bmatrix} \sum_i \frac{\partial \text{loss}}{\partial o_{i1}} x_{i1} & \ldots & \sum_i \frac{\partial \text{loss}}{\partial o_{im}} x_{i1} \\ \vdots & \ddots & \vdots \\ \sum_i \frac{\partial \text{loss}}{\partial o_{i1}} x_{if} & \ldots & \sum_i \frac{\partial \text{loss}}{\partial o_{im}} x_{if} \end{bmatrix} = \mathbf{X}^T \frac{\partial \text{loss}}{\partial \mathbf{X}\mathbf{W}}$$

The scalar derivative with respect to x_{ik} is $\frac{\partial o_{ij}}{\partial x_{ik}} = w_{kj}$. Hence, $\frac{\partial \text{loss}}{\partial x_{ik}} = \sum_j \frac{\partial \text{loss}}{\partial o_{ij}} w_{kj}$. Thus, the backward for $\mathbf{X}$ is

$$\frac{\partial \text{loss}}{\partial \mathbf{X}\mathbf{W}} \frac{\partial \mathbf{X}\mathbf{W}}{\partial \mathbf{X}} = \begin{bmatrix} \sum_j \frac{\partial \text{loss}}{\partial o_{1j}} w_{1j} & \ldots & \sum_j \frac{\partial \text{loss}}{\partial o_{1j}} w_{fj} \\ \vdots & \ddots & \vdots \\ \sum_i \frac{\partial \text{loss}}{\partial o_{nj}} w_{1j} & \ldots & \sum_i \frac{\partial \text{loss}}{\partial o_{nj}} w_{fj} \end{bmatrix} = \frac{\partial \text{loss}}{\partial \mathbf{X}\mathbf{W}} \mathbf{W}^T$$

question 4 The function $f(x)$ can be expressed as $f(x) = x\mathbf{1}$ where $\mathbf{1}$ is a 1×16 matrix with all values equal to 1. In that case, the local derivative can be defined as $\frac{\partial y_{ij}}{x_k} = \begin{cases} 1 & \text{if } k = i \\ 0 & \text{otherwise} \end{cases}$. Then using the multivariate chain rule, $\frac{\partial \text{loss}}{\partial x_i} = \sum_j \frac{\partial \text{loss}}{\partial y_{ij}}$. Therefore, the backward for $f(x)$ is $\frac{\partial \text{loss}}{\partial f(x)} \frac{\partial f(x)}{\partial x} = \frac{\partial \text{loss}}{\partial f(x)} \mathbf{1}^T$.

question 5

1) `c.value` contains the result of the element-wise addition between the values of `a` and `b`.
2) `c.source` points to the addition operator node.
3) `c.source.inputs[0]` points to the TensorNode `a`, meaning that `a is c.source.inputs[0]` returns `True`.
4) `a.grad` refers to the gradient with respect to `a`, $\frac{\partial \text{loss}}{\partial a}$, which is calculated during a backward call. Since backward has not been called yet, its values are all 0.

question 6

1) An operation is defined by the `Op` class or its subclasses but is not instantiated.
2) The actual addition happens on the return line of the forward method below.

```
class Add(Op):
    """
    Op for element-wise matrix addition.
    """
    @staticmethod
    def forward(context, a, b):
        assert a.shape == b.shape,\
            f'Arrays not the same sizes ({a.shape} {b.shape}).'
        return a + b
```

3) The output is not referenced at initialization because the `OpNode` is passed as the source reference to the outputs first. Once the outputs save the aforementioned node as the source, the reference to the outputs is saved in the `OpNode` object as well. This happens in the `do_forward` method of the `Op` class as seen below.

```
opnode = OpNode(cls, context, inputs)

outputs = [TensorNode(value=output, source=opnode)
           kfor output in outputs_raw]
opnode.outputs = outputs
```

question 7 The backward method in the `OpNode` calls the backward of an operation.

```
def backward(self):
    """..."""
    # extract the gradients over the outputs (these have been computed already)
    goutputs_raw = [output.grad for output in self.outputs]

    # compute the gradients over the inputs
    ginputs_raw = self.op.backward(self.context, *goutputs_raw)
```

question 8 Normalizing consists of transforming each element in $\mathbf{X}$ as such $y_{ij} = \frac{x_{ij}}{S_i}$, where $S_i = \sum_k x_{ik}$. Thus, the derivative is $\frac{\partial y_{ij}}{\partial x_{ik}} = \begin{cases} \frac{S_i - x_{ik}}{S_i^2} & \text{if } k = j \\ -\frac{x_{ik}}{S_i^2} & \text{otherwise} \end{cases}$.

This can be expressed as $\frac{\delta_{k=j}}{S_i} - \frac{x_{ik}}{S_i^2}$, where $\delta_{k=j} = \begin{cases} 1 & \text{k=j} \\ 0 & \text{otherwise} \end{cases}$. Then, using the multivariate chain rule

$$\frac{\partial loss}{\partial x_{ik}} = \sum_j \frac{\partial loss}{\partial y_{ij}} (\frac{\delta_{k=j}}{S_i} - \frac{x_{ik}}{S_i^2}) = \frac{\partial loss}{\partial y_{ik}} \frac{1}{S_i} - \frac{x_{ik}}{S_i^2} \sum_j \frac{\partial loss}{\partial y_{ij}} \Rightarrow$$

$$\begin{bmatrix} \frac{\partial loss}{\partial y_{11}} \frac{1}{S_1} - \frac{x_{11}}{S_1^2} \sum_j \frac{\partial loss}{\partial y_{1j}} & \cdots & \frac{\partial loss}{\partial y_{1m}} \frac{1}{S_1} - \frac{x_{1m}}{S_1^2} \sum_j \frac{\partial loss}{\partial y_{1j}} \\ \vdots & \ddots & \vdots \\ \frac{\partial loss}{\partial y_{n1}} \frac{1}{S_n} - \frac{x_{n1}}{S_n^2} \sum_j \frac{\partial loss}{\partial y_{nj}} & \cdots & \frac{\partial loss}{\partial y_{nm}} \frac{1}{S_n} - \frac{x_{nm}}{S_n^2} \sum_j \frac{\partial loss}{\partial y_{nj}} \end{bmatrix} =$$

$$\frac{\partial loss}{\partial \mathbf{Y}} \otimes \frac{1}{\mathbf{S}} - \frac{\mathbf{X}}{\mathbf{S}^2} \otimes \begin{bmatrix} \sum_j \frac{\partial loss}{\partial y_{1j}} & \cdots & \sum_j \frac{\partial loss}{\partial y_{1j}} \\ \vdots & \ddots & \vdots \\ \sum_j \frac{\partial loss}{\partial y_{nj}} & \cdots & \sum_j \frac{\partial loss}{\partial y_{nj}} \end{bmatrix} =$$

$$\frac{\partial loss}{\partial \mathbf{Y}} \otimes \frac{1}{\mathbf{S}} - \frac{\mathbf{X}}{\mathbf{S}^2} \otimes \frac{\partial loss}{\mathbf{Y}} \mathbf{1}$$

Here $\mathbf{S}$ is a $n \times m$ matrix with $\forall j \in 1, ..., m$ $S_{ij} = S_i$ and $\mathbf{1}$ is an $m \times m$ matrix with every value set to 1. The operations with respect to $\mathbf{S}$ are done element-wise. This is expressed a little differently than in code but is semantically equivalent.

question 9 The ReLU function can be seen below. The final validation accuracy it achieved was $\sim 95.6\%$ on the MNIST dataset. This is similar compared to the $\sim 97.2\%$ achieved by the same network with a sigmoid activation. The stochastic nature of neural networks would require to do multiple runs to make the comparison significant but the result is expected. Sigmoid and ReLU only begin to noticeably differ in performance on deep networks where sigmoid suffers from vanishing gradients.

```
class ReLU(Op):
    """
    Op for element-wise application of ReLU function
    """

    @staticmethod
    def forward(context, input):
        ind = input < 0
        context['ind'] = ind
        input[ind] = 0
        return input

    @staticmethod
    def backward(context, goutput):
        ind = context['ind']
        goutput[ind] = 0
        return goutput
```

question 10 I changed the MLP code to be able to handle a variable number of hidden layers as shown below.

```
def __init__(self, input_size, output_size, hidden_sizes=[8]):
    """
    :param hidden_sizes: the number of neurons in hidden layers
    """
    super().__init__()

    if len(hidden_sizes) > 0:
        self.initialize_layers(input_size, output_size,
                               hidden_sizes)
    else:
        self.layers = [vg.Linear(input_size, output_size),
                       vg.logsoftmax]

def initialize_layers(self, input_size, output_size, hidden_sizes):
    self.layers = [vg.Linear(input_size, hidden_sizes[0]), relu]
    for i in range(len(hidden_sizes[:-1])):
        self.layers.append(vg.Linear(hidden_sizes[i],
                                     hidden_sizes[i+1]))
        self.layers.append(relu)
    self.layers.extend([vg.Linear(hidden_sizes[-1], output_size),
                        vg.logsoftmax])

def forward(self, x):
    assert len(x.size()) == 2
    for l in self.layers: x = l(x)
    return x

def parameters(self):
    return [p for l in self.layers if hasattr(l, 'parameters')
            for p in l.parameters()]
```

I also added momentum:

```
momentum = [np.zeros_like(p.value) for p in mlp.parameters()]
...
for i, parm in enumerate(mlp.parameters()):
    momentum[i] = gamma * momentum[i] + parm.grad
    parm.value -= args.lr * momentum[i]
```

I then tried several different hyperparameters and settled on 3 hidden layers with 1568, 784, and 392 hidden neurons respectively. The learning rate was set to $3 \cdot 10^{-6}$ and the batch size was left as 128. For momentum, $\gamma = 0.9$. The behavior can be seen in figure 1. The final accuracy was $\sim 97.7\%$.

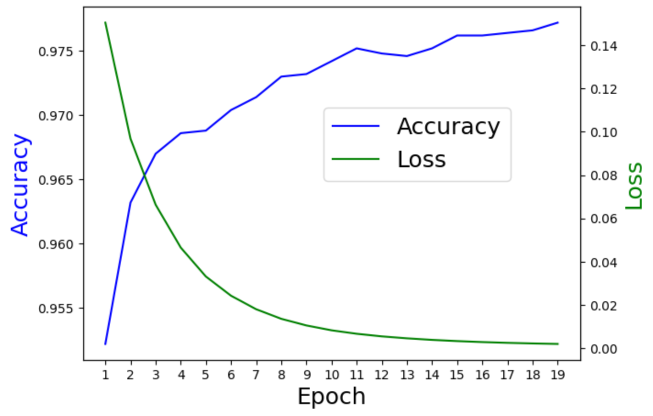
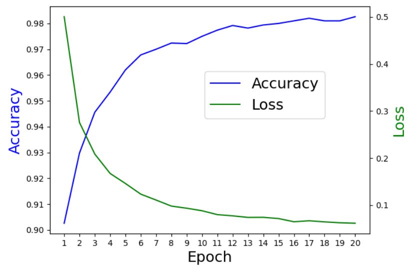


Figure 1: The performance with respect to validation loss and accuracy after every epochs of training on the MNIST dataset.

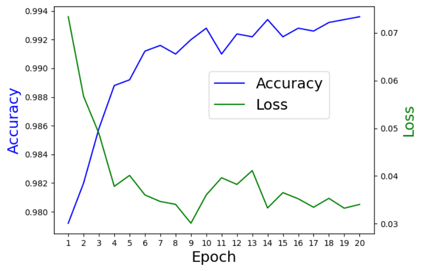
question 11 The best validation accuracy I achieved with a MLP implemented in PyTorch was $\sim 98.1\%$. The architecture and the training loop can be seen below. It has three hidden layers followed by dropout and ReLU activation layers. The dropout layers help with generalization. Moreover, the stochastic gradient descent was used with l2 regularization weighted as 10^{-4}. Momentum was also used with $\gamma = 0.9$. The learning rate and batch size hyperparamters were set to 10^{-3} and 32 respectively. Finally, the MNIST dataset was normalized to the range $[0, 1]$. Figure 2a shows the performance of the network after each epoch of training.

```
Sequential(
  (0): Linear(in_features=784, out_features=4096, bias=True)
  (1): Dropout(p=0.3, inplace=False)
  (2): ReLU()
  (3): Linear(in_features=4096, out_features=1024, bias=True)
  (4): Dropout(p=0.3, inplace=False)
  (5): ReLU()
  (6): Linear(in_features=1024, out_features=128, bias=True)
  (7): Dropout(p=0.3, inplace=False)
  (8): ReLU()
  (9): Linear(in_features=128, out_features=10, bias=True)
  (10): Dropout(p=0.3, inplace=False)
  (11): LogSoftmax(dim=1)
)

EPOCH = 20
criterion = nn.NLLLoss()
```

```
results = list()

# Training loop
for e in range(EPOCH):
    results.append(evaluate(model, e=e))
    train(model)
results.append(evaluate(model))

def evaluate(model, e=EPOCH):
    model.eval()
    out = model(xval)
    loss = criterion(out, yval)
    acc = float((yval == out.argmax(dim=1)).to(float).mean())

    print(f"\n-------------EPOCH {e}-------------")
    print(f"Validation accuracy: {acc:.4f}\tLoss: {loss:.4f}")

    return {e: {'loss': loss, 'acc': acc}}

def train(model):
    model.train()
    for x, y in tqdm(train_loader):
        opt.zero_grad()
        loss = criterion(model(x), y)
        loss.backward()
        opt.step()
```



(a) Performance of the MLP implemented in PyTorch



(b) Performance of the convolutional neural network

Figure 2

question 12 I attempted several techniques to improve the efficacy of the classifier neural network. The first of which is using convolutional layers which made the model train faster and led to a better overall performance. I also switched to the Adam optimizer with learning rate set to 10^{-4} and l2 regularization also set to 10^{-4}. This similarly had a positive effect over SGD although I did not complete multiple runs to make sure that the improvement is significant. After the convolutional layers, I kept a similar structure as before with dropout layers following linear layers to help with generalization. Final thing I attempted was to add batch normalization before the convolutional layers. This had a positive effect on getting a good solution quickly, after just one or two epochs, but did not appear to change the overall best model found by the end of training. The architecture can be seen below.

```
Sequential(
  (0): BatchNorm2d(1, eps=1e-05, momentum=0.1, ...)
  (1): Conv2d(1, 32, kernel_size=3, stride=1, padding=1)
  (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1)
  (3): Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
  (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1)
  (5): Conv2d(64, 128, kernel_size=3, stride=1, padding=1)
  (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1)
  (7): Flatten(start_dim=1, end_dim=-1)
  (8): Linear(in_features=1152, out_features=256, bias=True)
  (9): Dropout(p=0.3, inplace=False)
  (10): ReLU()
  (11): Linear(in_features=256, out_features=64, bias=True)
  (12): Dropout(p=0.3, inplace=False)
  (13): ReLU()
  (14): Linear(in_features=64, out_features=10, bias=True)
  (15): Dropout(p=0.3, inplace=False)
  (16): LogSoftmax(dim=1)
)
```

Figure 2b shows the performance after every epoch of training for the convolutional neural network. It is a bit noisier than the MLP but that is likely because it is performing with overall better accuracy where smaller deviations in the model space lead to greater drops in performance. The final accuracy achieved on the validation set was ~ 99.4%.