Deep Learning - Assignment 1

Carlos Vega and Marcos Alonso

Backpropagation

- **Code**:

```
def backpropagation(model, y_true, y_pred):
    with torch.no_grad():
        # Derivative of the loss with respect to the output of the network
        dL_da = 2*(y_pred - y_true)

        for layer in range(model.L, 0, -1):
            # Derivative of activation function

            # Derivative of activation function with respect to z
            da_dz = model.df[layer](model.z[layer])
            # Derivative of the loss with respect to z
            dL_dz = dL_da * da_dz

            if model.a[layer-1].ndimension() == 1:
                activations_prev_layer = model.a[layer-1].unsqueeze(0)
            else:
                activations_prev_layer = model.a[layer-1]

            # Derivative of the loss with respect to the weights
            dL_dw = torch.mm(dL_dz.T, activations_prev_layer)

            # Derivative of the loss with respect to the biases
            dL_db = dL_dz.sum(0)

            model.dL_dw[layer] = dL_dw
            model.dL_db[layer] = dL_db

            if layer > 1:
                # Derivative of the loss with respect to the output of the previous
layer
                dL_da = torch.mm(dL_dz, model.fc[str(layer)].weight).sum(0)
```

- **Explanation**:

We followed the formulas given in the theory to compute the algorithm. The explanations of each step are commented in the given code.

Gradient descent

We first loaded the database

Load the database

```
transform = transforms.Compose([
    transforms.ToTensor(),                          #converts to tensor
    transforms.Lambda(lambda x: torch.flatten(x))   #Flattens the tensor
])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                         download=True, transform=transform)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                        download=True, transform=transform)
```

Then filter by the categories that we were asked

Filter by category

```
classes_to_keep = ['bird', 'airplane']
class_indices = [trainset.class_to_idx[c] for c in classes_to_keep]
```
✓ 0.0s

```
subset_trainset = Subset(trainset, [i for i in range(len(trainset)) (variable) testset: Any class_indices])
subset_testset = Subset(testset, [i for i in range(len(testset)) if testset.targets[i] in class_indices])
```
✓ 0.0s

```
train_size = int(0.8 * len(subset_trainset))
val_size = len(subset_trainset) - train_size
trainset, valset = random_split(subset_trainset, [train_size, val_size])
```
✓ 0.0s

```
train_loader = DataLoader(trainset, batch_size=32)
valloader = DataLoader(valset, batch_size=32)
testloader = DataLoader(subset_testset, batch_size=32)
```
✓ 0.0s

Then we created the neural network with the layers specified

Create Neural Network

```python
class MyMLP(nn.Module):
    def __init__(self):
        super(MyMLP, self).__init__()
        self.fc1 = nn.Linear(3072, 512)  # Fully connected layer 1
        self.fc2 = nn.Linear(512, 128)   # Fully connected layer 2
        self.fc3 = nn.Linear(128, 32)    # Fully connected layer 3
        self.fc4 = nn.Linear(32, 2)      # Fully connected  output layer

    def forward(self, x):
        # Forward propagation via ReLU
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = torch.relu(self.fc3(x))
        x = self.fc4(x)  # No activation function at the output
        return x
```

The train function

Train function

```python
def train(n_epochs, optimizer, model, loss_fn, train_loader):
    for epoch in range(1, n_epochs+1):
        running_loss = 0.0
        for inputs, labels in train_loader:
            optimizer.zero_grad()  # Reset gradients in each iteration

            outputs = model(inputs)  # Forward propagation
            labels = torch.Tensor([[1.0, 0.0] if i.item() == 0 else [0.0, 1.0] for i in labels])
            loss = loss_fn(outputs, labels)  # Loss calculation

            loss.backward()  # Back propagation
            optimizer.step()  # Update parameters

            running_loss += loss.item() * inputs.size(0)

        epoch_loss = running_loss / len(train_loader.dataset)
        print(f'Epoch [{epoch}/{n_epochs}], Loss: {epoch_loss:.4f}')
```

And then we created the Manual train function. We made the function return the final loss parameter in order to be able to compare afterwards which approach gets the best loss. We did it to generalize the code and make it more flexible for comparing.

Manual Train Function

The function returns the last loss to be able to afterwards compare the results

```python
def train_manual_update(n_epochs, model, loss_fn, train_loader, lr, weight_decay=0.0, momentum=0.0):
    velocities ={i: 0 for i, p in enumerate(model.parameters())}
    for epoch in range(1, n_epochs+1):
        running_loss = 0.0
        for inputs, labels in train_loader:
            model.train()  # Make sure the model is in train
            outputs = model(inputs)  # Forward Propagation

            labels = torch.Tensor([[1.0, 0.0] if i.item() == 0 else [0.0, 1.0] for i in labels])
            loss = loss_fn(outputs, labels)  # Loss Calculation

            # Calculate gradients manualy
            loss.backward()

            # Update parameters manually using learning rate
            with torch.no_grad():
                for i, param in enumerate(model.parameters()):
                    gradient = param.grad

                    if weight_decay != 0:
                        gradient = gradient.add_(param.data, alpha=weight_decay)

                    if momentum != 0:
                        velocities[i] = velocities[i] * momentum + gradient
                        gradient = velocities[i]

                    new_param = param.data.add_(gradient, alpha=-lr)
                    param.copy_(new_param)
                    param.grad.zero_()

            # Reset gradients
            model.zero_grad()

            running_loss += loss.item() * inputs.size(0)

        # Average loss per epoch
        epoch_loss = running_loss / len(train_loader.dataset)
        print(f'Epoch [{epoch}/{n_epochs}], Loss: {epoch_loss:.4f}')

    # We return the final loss value in able to afterwards know the best aproach
    return epoch_loss
```

We proceed to train both models.

Train both models

```
n_epochs = 10
lr = 0.001

model = MyMLP()
model_manual = MyMLP()

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=lr)

train(n_epochs, optimizer, model, criterion, train_loader)
train_manual_update(n_epochs, model_manual, criterion, train_loader, lr, weight_decay=0.1, momentum=0.9)
```

✓ 45.4s

And evaluate the results

```
model.eval()
model_manual.eval()

epoch_loss_train = 0.0
epoch_loss_manual = 0.0

with torch.no_grad():
    for inputs, labels in train_loader:
        outputs = model(inputs)

        labels = torch.Tensor([[1.0, 0.0] if i.item() == 0 else [0.0, 1.0] for i in labels])
        loss = criterion(outputs, labels)
        epoch_loss_train += loss.item() * inputs.size(0)

        outputs_manual = model_manual(inputs)
        loss_manual = criterion(outputs_manual, labels)
        epoch_loss_manual += loss_manual.item() * inputs.size(0)

epoch_loss_train /= len(train_loader.dataset)
epoch_loss_manual /= len(train_loader.dataset)

# Print loss for each model
print("Loss (train):", epoch_loss_train)
print("Loss (train_manual_update):", epoch_loss_manual)
```

✓ 1.4s

```
Loss (train): 0.6526539619248303
Loss (train_manual_update): 0.5233056174851183
```

Now we train the manual train with some different parameters

Train with different instances

```
# Train 4 models with different learning rates, momentum and weight decay
models = []
for _ in range(4):
    model = MyMLP()
    models.append(model)

n_epochs = 10
criterion = nn.CrossEntropyLoss()
performance = []
performance.append(train_manual_update(n_epochs, models[0], criterion, train_loader, lr=0.001, weight_decay=0.0, momentum=0.0))
performance.append(train_manual_update(n_epochs, models[1], (variable) criterion: Any =0.01, weight_decay=0.0, momentum=0.0))
performance.append(train_manual_update(n_epochs, models[2],                          =0.001, weight_decay=0.1, momentum=0.9))
performance.append(train_manual_update(n_epochs, models[3], criterion, train_loader, lr=0.01, weight_decay=0.0, momentum=0.9))
```

Then we analyze the one that provided the best results

Best performance model

```
# Get the best perfomance
bestModelIndex = performance.index(min(performance))
print("The model", bestModelIndex+1, "had the best approach")
bestModel = models[bestModelIndex]

# Model 4 has the best performance
# Model 4 on unseen data

bestModel.eval()

epoch_loss_test = 0.0

with torch.no_grad():
    for inputs, labels in testloader:
        outputs = bestModel(inputs)

        labels = torch.Tensor([[1.0, 0.0] if i.item() == 0 else [0.0, 1.0] for i in labels])
        loss = criterion(outputs, labels)
        epoch_loss_test += loss.item() * inputs.size(0)

epoch_loss_test /= len(testloader.dataset)

print("Loss validation (test):", epoch_loss_test)
```

✓ 0.3s

```
The model 4 had the best approach
Loss validation (test): 0.41951453106993797
```

And the best model between the four selected was the one with these parameters: lr=0.01, weight_decay=0.0, momentum=0.9

Questions

a) **Which PyTorch method(s) correspond to the tasks described in section 2?**

In section 2 we do all the calculations manually but the pytorch method that is used to compute the backpropagation automatically is loss.backward()

b) **Cite a method used to check whether the computed gradient of a function seems correct. Briefly explain how you would use this method to check your computed gradients in section 2.**
Pytorch provides a method autograd that can check that the gradients are correct.
It is also possible to do it mathematically following the formula:
(f(x+h) - f(x-h))/2h

c) **Which PyTorch method(s) correspond to the tasks described in section 3, question 4.?**

In pytorch it is equivalent to manually modify the parameters of the model. Pytorch uses optimizer in order to do so.

d) **Briefly explain the purpose of adding momentum to the gradient descent algorithm**

Adding momentum accelerates convergence and it improves stability by incorporating past gradients into updates, reducing oscillations and potentially escaping local minima faster.

e) **Briefly explain the purpose of adding regularization to the gradient descent algorithm.**

The main advantage it provides is to avoid overfitting, making a more general answer that can work better with unseen data

f) **Report the different parameters used in section 3, question 8., the selected parameters in question 9. as well as the evaluation of your selected model.**

These are the values we selected and we could see clearly which approach had the best result.

Learning Rate	Weight Decay	Momentum	Final Loss
0.001	0	0	0.6469
0.01	0	0	0.4616
0.001	0.1	0.9	0.5427
0.01	0	0.9	**0.4162**

g) **Comment your results. In case you do not get expected results, try to give potential reasons that would explain why your code does not work and/or your results differ.**

The results make sense as the learning rate 0.01 is much better than 0.001 because 0.001 seems to be too small that the updates are not very significant. The momentum helps to get out of local minimums and therefore makes sense that improves the results.