Report Project 1 - igu011 and edj001

Division of labor

We (igu011 and edj001) have worked closely through the whole project. We pair programmed on both files. igu011 worked on the report while edj001 implemented the last part of gradient_descent.ipynb where we check accuracies and evaluate the best model.

1

backpropagation.ipynb

To manually implement the optimization step there is no need for many lines of code. We translated the given formulas to python and PyTorch.

```
dz_prev = None
for l in range (1, model.L + 1) [::-1]:
 if l == model.L:
 # Output layer
 dz_l = (-2 * (y_true - y_pred)) * model.df[l](model.z[l])
 else:
 # Hidden layers
 dz_l = torch.mm(dz_prev, model.fc[str(l+1)].weight.data) * model.df[l]
(model.z[l])

 dw_l = (1/batch_size) * torch.mm(dz_l.t(), model.a[l-1])
 db_l = (1/batch_size) * torch.sum(dz_l, dim=0, keepdims=True)
 db_l = torch.flatten(db_l)
```

gradient_descent.ipynb

L2 reguralization

We see that the results differ some with the manual implementation of L2 regularization. We suspect that this is from the L2 loss formula which I have struggled a bit with. We initially thought from Andrew's videos was that we could implement with `L2_reg += torch.pow(param, 2).sum()` or `L2_reg += param.norm(2)`, but with this implementation, we got drastically different results from the PyTorch implementation. We saw that often the formula is expressed as `/2 * w **2` so we tried with the implementation `0.5 * torch.pow(param, 2).sum()` which gave us more similar results. The current implementation works a bit simpler by using the fact that parameters have to be loaded and iterated over later during corrections performed by the optimizer. With this in mind, we don't need to do the power of 2 because the gradient of w*2 *is* 2w. We modify the existing gradient by adding p.data (weight) multiplied by weight_decay which results in an implementation done in place. The last line updates the weights with the gradient with the standard SGD formula.

Momentum

We implemented momentum by the definition given in Andrew's videos. We initialize tensor of velocities which are computed for every gradient using the momentum input. With our manual momentum implementation, we see that the two implementations converge in the same matter, but have slightly different numbers. We cannot quite pinpoint what makes these small differences, since to our understanding the momentum algorithm is implemented with the same logic as PyTorch's implementation.

2

a)

Formula 5 from section 2 is implemented with the following code:

```
dz_l = (-2 * (y_true - y_pred)) * model.df[l](model.z[l])
```

Formula 6 from section 2 is implemented with the following code:

```
dz_l = torch.mm(dz_prev, model.fc[str(l+1)].weight.data) * model.df[l]
(model.z[l])
```

Formula 4 from section 2 is implemented with the following code:

```
dw_l = (1/batch_size) * torch.mm(dz_l.t(), model.a[l-1])
db_l = (1/batch_size) * torch.sum(dz_l, dim=0, keepdims=True)
db_l = torch.flatten(db_l)
```

With a pure pytorch implementation we would use `loss.backward()` and `optimizer.step()`

b)

To check whether the computed gradient of a function seems correct in pytorch one could use `relative_error(weight, dL_dw)` to see the difference in weights and `relative_error(bias, dL_db)` where relative error can be computed by `torch.norm(a - b) / torch.norm(a)`

c)

Using PyTorch we specify the learning rate when we initialize the optimizer.

```
optimizer = optim.SGD(model.parameters(), lr=0.01)
```

d)

Momentum is an extension of the gradient descent optimization algorithm. It allows the search to build inertia in a direction in the search space and overcome the oscillations of noisy gradients and cost across

flat sports of the search space. The momentum is designed to accelerate the optimization process. One problem with the gradient descent algorithm is that the progression of the search can bounce around the search space based on the gradient. This can slow down the process of the search, especially for those optimization problems where the broader trend or shape of the search space is more useful than specific gradients along the way.

Momentum involves adding a hyperparameter that controls the amount of history (momentum) to include in the update equation. The value of a hyperparameter is defined in the range 0.0 to 0.1 and often has a value close to 1.0, i.e. 0.9. A momentum of 0.0 is the same as gradient descent without momentum.

e)

Regularization is a technique used to reduce errors by fitting the function appropriately on the given training set and avoiding overfitting.

The commonly used regularization techniques are L1 regularization, L2 regularization, and Dropout regularization. In our code, we use L2 regularization and the L2 regularization is also called Ridge Regression. L2regularization uses "squared magnitude" of coefficient as penalty term of the loss function.

f)

Global parameters

batch_size = 256
n_epoch = 30
loss_fn = nn.CrossEntropyLoss()
seed = 265

Model 1

lr=0.01
weight_decay=0
momentum=0

Accuracy train: 0.88
Accuracy val: 0.83

Model 2

lr=0.01
weight_decay=0.01
momentum=0

Accuracy train: 0.86
Accuracy val: 0.83

Model 3

lr=0.01
weight_decay=0
momentum=0.9

Accuracy train: 0.90
Accuracy val: 0.81

Model 4

lr=0.01
weight_decay=0.001
momentum=0.9

Accuracy train: 0.96
Accuracy val: 0.84
Accuracy test: 0.85

Model 5

lr=0.02
weight_decay=0.01
momentum=0.8

Accuracy train: 0.88
Accuracy val: 0.81

g)

We get expected results from our model with almost the same accuracy for validation (84%) and test (85%). This means we can conclude that our model will generalize well, by correctly labeling 85% of unseen images of birds or planes.