INF265 Project 1 Report

Ninad Hagi

21 February 2025

1 Explanation of Implementation and Design Choices

1.1 Backpropagation

Network
Used a custom neural network class `MyNet`, which stores the necessary tensors for each layer ($z^{[l]}$, $a^{[l]}$) and provides a forward pass.

- **Manual Gradient Computation:**
  - Implemented `backpropagation(model, y_true, y_pred)` to compute $\frac{\partial L}{\partial w}$ and $\frac{\partial L}{\partial b}$ in a manner consistent with the equations given:

$$\delta^{[L]} = \frac{\partial L}{\partial \hat{y}} \cdot f'^{[L]}(z^{[L]}), \quad \delta_i^{[l]} = \left(\sum_k \delta_k^{[l+1]} \, w_{k,i}^{[l+1]} \right) \times f'^{[l]}\big(z_i^{[l]}\big).$$

  - Then stored the gradients in `model.dL_dw[l]` and `model.dL_db[l]`.
- **Loss Function:**
  - Used **Mean Squared Error (MSE)** for the backprop test.
  - Our derivative of the MSE is

$$\nabla_{y_{pred}} L = -2(y_{true} - y_{pred}).$$

- **Backprop Algorithm:**
  - Output-layer local gradient:

$$\delta^{[L]} = \frac{\partial L}{\partial y_{pred}} \times f'^{[L]}(z^{[L]})$$

  - Hidden-layer local gradient:

$$\delta_i^{[l]} = \left(\sum_{k=1}^{n^{[l+1]}} \delta_k^{[l+1]} w_{k,i}^{[l+1]} \right) \times f'^{[l]}(z_i^{[l]})$$

  - We store:
$$\frac{\partial L}{\partial w_{i,j}^{[l]}} = \delta_i^{[l]} \times a_j^{[l-1]} \quad and \quad \frac{\partial L}{\partial b_j^{[l]}} = \delta_j^{[l]}.$$

- **Validation via Autograd & Gradient Checking:**
  - Confirmed correctness by comparing:
    1. Our manual gradients vs. PyTorch Autograd (`loss.backward()`).
    2. Our manual gradients vs. finite-differences (gradient checking).

1.2 Backpropagation Output

```
    ------------------------------------------------------------------
                Check gradients
------------------------------------------------------------------

====================== Epoch 1 ======================

-------- Gradcheck with finite differences  ---------
residual error:
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

--------- Comparing with autograd values  -----------

******* fc['1'].weight.grad *******
Our computation:
tensor([[ 6.7814e-08,  7.2355e-08],
        [-1.0720e-03, -1.1438e-03],
        [-2.2284e-03, -2.3776e-03]])

Autograd's computation:
tensor([[ 6.7814e-08,  7.2355e-08],
        [-1.0720e-03, -1.1438e-03],
        [-2.2284e-03, -2.3776e-03]])

...

Conclusion:
4 / 4: ALL TEST PASSED :)
```

The computed gradients match perfectly `Residual error:  0.0`.

2 Gradient descent

- **Data (CIFAR-10):**

  - Kept only two classes: "cat" and "car," resized images to 16×16, and split the dataset into training/validation/test sets (with `shuffle=False` to ensure determinism).

- Used two approaches to update parameters:
  1. **Built-in PyTorch Optimizer** (`torch.optim.SGD`).
  2. **Manual Updates** (by explicitly applying the weight-update equation).

- **Network:**
  - A small MLP (`MyMLP`) with input dimension = 768, two hidden layers (128 and 32 units), ReLU activation, and 2 output units.

- **Model:**
  - A small MLP (`MyMLP`) with input size 768 ($16 \times 16 \times 3$), two hidden layers of sizes [128, 32], and output size = 2.
  - All activations are ReLU, except no activation on the final layer.

- **Training Functions:**
  1. `train(...)` using `torch.optim.SGD(model.parameters(), ...)`.
  2. `train_manual_update(...)` that manually updates each parameter as:
$$p \leftarrow p - \alpha\big(\nabla_p L + \lambda\, p\big)$$

- **Training Routines:**
  1. `train(...)`: Uses a standard `torch.optim.SGD`.
  2. `train_manual_update(...)`: Manually updates parameters by
$$\theta \leftarrow \theta \;-\; \alpha\big(\nabla_\theta L \;+\; \texttt{weight\_decay} \times \theta\big),$$

- **Verification of identical results:**
  - Set the same random seeds (`torch.manual_seed(...)`), the same data ordering (`shuffle=False`), and used double precision (`torch.set_default_dtype(torch.doub`

- **Ensuring exact match:**
  - To match the results of `train(...)` vs. `train_manual_update(...)`, we:
    * Set `torch.manual_seed(42)` for both.
    * Used `torch.set_default_dtype(torch.double)`.
    * Disabled `shuffle` in the `DataLoader`.
    * Converted inputs to `dtype=torch.double`.

2.1 Gradient Descent output (sample)

```
    Training on device cpu.

Global parameters:
batch_size = 256
n_epoch = 30
loss_fn = CrossEntropyLoss()
seed = 265

The best model was trained with
    lr = 0.01
    mom = 0
    decay = 0
Training accuracy of the best model:
Accuracy: 0.97
Validation accuracy of the best model:
Accuracy: 0.91

Test Accuracy score of the best model
Accuracy: 0.91
```

Training Loss Logs (using Pytorch´s SGD)

```
Epoch 1  |  Training loss 0.67882
Epoch 5  |  Training loss 0.56008
Epoch 10 |  Training loss 0.43988
Epoch 15 |  Training loss 0.37431
Epoch 20 |  Training loss 0.32573
Epoch 25 |  Training loss 0.29081
Epoch 30 |  Training loss 0.26554
```

In addition, I also trained a model using `train_manual_update()` under the same hyperparameters (LR = 0.01, momentum = 0, decay = 0) and observed that the final results match within floating-point tolerance.

3 Questions

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

Section 2 involves *backpropagation* to compute $\frac{\partial L}{\partial w}$. In PyTorch, one can typically rely on:

- `loss.backward()` to compute gradients automatically,

- `p.grad` to access the gradient of parameter `p` after `loss.backward()` is called.

Hence, the PyTorch methods that do the job I implemented manually in `backpropagation(...)` are the *automatic differentiation* routines built into PyTorch Autograd. Specifically:

- `torch.autograd.backward(...)` or equivalently calling `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.

A standard approach is **gradient checking** using **finite differences**:

1. For each parameter p, you slightly perturb it by ϵ,
2. Evaluate the loss at $p + \epsilon$ and $p - \epsilon$,
3. Approximate the gradient as
$$\frac{L(p+\epsilon) - L(p-\epsilon)}{2\epsilon}.$$
4. Compare that approximation to your backprop-computed gradient.

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

Section 3, question 4 asks us to *manually update each trainable parameter* based on the computed gradient. The **built-in** equivalent is:

- `torch.optim.SGD` (with possible arguments for momentum, weight decay, etc.).
- The line `optimizer.step()` specifically updates the parameters under the hood using the gradients stored in `p.grad`.

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

Momentum helps accelerate training by **accumulating** a velocity vector. Instead of doing:

$$\theta \leftarrow \theta - \alpha \nabla_\theta L,$$

Perform:

$$v \leftarrow \mu v + \nabla_\theta L, \quad \theta \leftarrow \theta - \alpha v,$$

where μ is a momentum coefficient in $(0, 1)$. This reduces oscillations in directions of small but persistent gradients, and speeds up convergence along consistent gradient directions.

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

Regularization (L2 weight decay) aims to **penalize large weights** and thus reduce overfitting. It effectively shrinks parameters over time by adding $\lambda\theta$ to the gradient. Concretely:

$$\theta \leftarrow \theta - \alpha\big(\nabla_\theta L(\theta) + \lambda\theta\big).$$

(f) Hyperparameters, Selected Model, and Evaluation

Tested a few hyperparameter combinations for learning rate (`lr`), momentum (`mom`), and weight decay (`decay`), but the final best model was:

- **lr** = 0.01
- **momentum** = 0
- **weight decay** = 0

Under these settings (and batch size = 256, 30 epochs), we obtained:

- **Training Accuracy**: 97%
- **Validation Accuracy**: 91%
- **Test Accuracy**: 91%

The model trained with `torch.optim.SGD` and the model trained via our manual updates gave nearly identical performance when using the same seed, data ordering, learning rate, etc.

(g) Discussion of Results

1. **Overall Performance**
    - A training accuracy of 97% and validation/test accuracy of 91% indicates our MLP can effectively distinguish cars vs. cats in the reduced CIFAR-10 dataset.
    - The $\sim$6% gap between training and validation suggests some overfitting, but not excessively so.

2. **Why No Momentum or Weight Decay May Work Well**

    - For this two-class subset, a straightforward SGD approach with a moderate learning rate of 0.01 seems sufficient. More complex hyperparameters (momentum, decay) did not outperform the simpler setting in our trials.

3. **Verification of Manual vs. Built-in Training**

    - Confirmed that our manual gradient descent updates produce the same final losses and accuracies as `torch.optim.SGD` when using the same initialization (seed), data ordering (`shuffle=False`), learning rate, and so on.

4. **Backpropagation Checks**

    - The backpropagation code passed the gradient-checking tests with a residual error of 0.0, indicating that our manually computed gradients precisely match both finite-differences and PyTorch's autograd. This strongly validates the correctness of our Section 2 implementation.

5. **Possible Variations**

    - Different seeds or data splits might change accuracies slightly.
    - On a larger or more complex classification task, momentum and L2 regularization might help more.

Overall, our experiments confirm that **(1)** our backprop implementation is accurate, **(2)** our manual updates align with PyTorch's built-in SGD, and **(3)** our best model achieves 91% accuracy on distinguishing cats vs. cars.

Conclusion

Overall, both our **Backpropagation** and **Gradient Descent** implementations are correct. I confirmed that the manual gradients are accurate (finite-differences residual error is 0.0), and our manually updated model's performance aligns with PyTorch's built-in `SGD` when identical hyperparameters and seeds are used. Our final test accuracy of about 91% is solid for a 2-class subset of CIFAR-10.