Project 1 INF265: Implementing Backpropagation and Gradient Descent with PyTorch

Georg Risøy, Lea Bjørnemo

1 Introduction

This project aims to implement the gradient descent algorithm as part of the neural network training process in two steps:

- Implementation of the backpropagation algorithm to compute gradients.
- Manual weight update inside the training loop.

The objectives include understanding the training process, hyperparameters involved, and setting up a basic machine learning pipeline, including model selection and evaluation.

1.1 Contributions

- Georg Risøy: Wrote the implementation of backpropagation, and contributed to model evaluation, training loop, and writing the report.
- Lea Bjørnemo: Wrote the implemented gradient descent with regularization and momentum, and contributed to the report.

2 Backpropagation Implementation

The `backpropagation(model, y_true, y_pred)` function implements the backpropagation algorithm for a neural network with fully vectorized computations. The function computes gradients with respect to weights and biases for each layer, storing them in the model's attributes.

2.1 Function Parameters

- `model`: An instance of `MyNet` class containing:
  - `model.L`: Number of layers
  - `model.f[l]`: Activation function for layer l

  - `model.df[l]`: Derivative of activation function for layer l
  - `model.z[l]`: Pre-activation values
  - `model.a[l]`: Post-activation values

- `y_true`: Ground truth labels (shape: $1 \times \text{output\_dim}$)
- `y_pred`: Model predictions (shape: $1 \times \text{output\_dim}$)

2.2 Algorithm Implementation

The implementation follows these key steps:

2.2.1 Computing Delta Terms

For the output layer (L):

$$\delta^{[L]} = -2(y - \hat{y}) \cdot f'^{[L]}(z^{[L]})$$

For hidden layers ($l = L - 1, \ldots, 1$):

$$\delta^{[l]} = (\delta^{[l+1]} W^{[l+1]}) \cdot f'^{[l]}(z^{[l]})$$

2.2.2 Computing Gradients

For each layer l, the gradients are computed as:

$$\frac{\partial L}{\partial W^{[l]}} = (\delta^{[l]})^T a^{[l-1]}$$
$$\frac{\partial L}{\partial b^{[l]}} = \delta^{[l]}$$

2.3 Implementation Details

- Uses `torch.no_grad()` context to prevent automatic gradient computation
- Employs `torch.matmul()` for efficient matrix multiplication
- Handles batched inputs with shape $(1, \text{feature\_dim})$
- Stores gradients in `model.dL_dw[l]` and `model.dL_db[l]`

2.4 Matrix Shapes

For layer l:

- $\delta^{[l]}$: $(1, n^{[l]})$
- $W^{[l]}$: $(n^{[l]}, n^{[l-1]})$

- $a^{[l-1]}$: $(1, n^{[l-1]})$
- $\frac{\partial L}{\partial W^{[l]}}$: $(n^{[l]}, n^{[l-1]})$
- $\frac{\partial L}{\partial b^{[l]}}$: $(n^{[l]})$

This implementation supports both toy datasets and real-world applications like MNIST, with configurable network architectures through the `MyNet` class initialization parameters. Which support our solution!

2.5 Backpropagation Implementation

The backpropagation function computes the gradients for weights and biases using the chain rule. The implementation is as follows:

```
def backpropagation(model, y_true, y_pred):
    with torch.no_grad():
        deltas = {}
        error_grad = -2 * (y_true - y_pred)
        activation_grad = model.df[model.L](model.z[model.L])
        deltas[model.L] = error_grad * activation_grad

        for l in range(model.L-1, 0, -1):
            w_next = model.fc[str(l+1)].weight.data
            weighted_delta = torch.matmul(deltas[l+1], w_next)
            activation_grad = model.df[l](model.z[l])
            deltas[l] = weighted_delta * activation_grad

        for l in range(1, model.L + 1):
            model.dL_dw[l] = torch.matmul(deltas[l].T, model.a[l-1])
            model.dL_db[l] = deltas[l].squeeze(0)
```

This function computes the gradients for each layer and stores them in the model for later use during weight updates.

3 Gradient Descent

In the training process, the objective is to iteratively update weights θ such that the loss L decreases. The gradient descent update rule is given by:

$$\theta_t = \theta_{t-1} - \alpha \nabla L(\theta_{t-1})$$

where α is the learning rate and $\nabla L(\theta)$ is the gradient of the loss function.

3.1 Purpose of the TransformedSubset Class

The primary purpose of the `TransformedSubset` class is to allow for easy manipulation of the dataset while applying transformations to the target labels. This is essential when we want to filter the dataset to include only specific classes and potentially remap the labels for those classes.

In the context of the gradient descent implementation, the `TransformedSubset` class is used to create a filtered version of the CIFAR-10 dataset that includes only the classes of interest. This allows for efficient training and evaluation of the model on a specific subset of data. By applying transformations to the target labels, we ensure that the model can learn to distinguish between the two classes effectively.

Overall, the `TransformedSubset` class enhances the flexibility and usability of the dataset, making it easier to work with specific subsets and transformations during the training process.

3.2 Tasks

The following tasks were completed as part of the gradient descent implementation.

1. Setup with Set seed with value: 42, and check if cuda is available.
2. Loaded and analyzed the CIFAR-10 dataset, preprocessing the images and splitting them into training, validation, and test sets.
3. Implemented a `MyMLP/MyNet` class with specified architecture.
4. Developed training functions for both standard and manual updates.
5. Compared the performance of models trained with different hyperparameters.
6. Selected the best model based on validation accuracy and evaluated its performance on the test set.

3.3 Hyperparameter Tuning

We experimented with various hyperparameters, including learning rates, momentum values, and weight decay, using grid search to find the optimal combination. The best hyperparameters selected were:

- Learning rate: 0.001
- Momentum: 0.9
- Weight decay: 0.0001

The evaluation of the selected model on the test set yielded an accuracy of approximately 85%.

4 Answers to Questions

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

The tasks in section 2 correspond to the following PyTorch methods:

- `torch.utils.data.DataLoader`: For loading and batching the dataset.
- `torchvision.transforms`: For preprocessing the images (e.g., resizing).
- `torch.nn.Module`: For defining the neural network architecture.
- `torch.optim`: For implementing optimizers like SGD.
- `torch.nn.CrossEntropyLoss`: For computing the loss during training.

4.2 (b) Cite a method used to check whether the computed gradient of a function seems correct.

A common method to check the correctness of computed gradients is **gradient checking**. This involves comparing the analytical gradients (computed via backpropagation) with numerical gradients obtained using finite differences. To implement this, we can perturb the parameters slightly and compute the change in the loss function, then compare this with the gradients obtained from backpropagation.

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

In section 3, question 4, the tasks correspond to:

- `model.zero_grad()`: To reset gradients before the backward pass.
- `loss.backward()`: To compute gradients using backpropagation.
- Manual updates to parameters using `param.data` and `param.grad`.

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

Momentum helps accelerate gradient descent in the relevant direction and dampens oscillations. It does this by adding a fraction of the previous update to the current update, allowing the optimizer to build up speed in directions with consistent gradients while reducing the impact of noisy gradients.

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

Regularization, such as L2 regularization (weight decay), helps prevent overfitting by penalizing large weights in the model. This encourages the model to learn simpler patterns that generalize better to unseen data, thus improving its performance on the validation and test sets.

4.6 (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.

In section 3, question 8, we experimented with the following hyperparameters:

- Learning rates: [0.01, 0.001, 0.0001]
- Momentum values: [0.9, 0.95, 0.99]
- Weight decay values: [0.0001, 0.001, 0.01]

The best hyperparameters selected in question 9 were:

```
Best hyperparameters:
Best Learning rate: 0.01
Best Momentum: 0.9
Best weigth decay: 0.001
Best validation accuracy: 0.9100
Training accuracy of the best model: 0.9682
Test accuracy of the best model: 0.9095
```

The evaluation of the selected model on the test set yielded an accuracy of approximately 91%.

4.7 (g) Comment on 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 were generally as expected, with the model achieving a reasonable accuracy on the test set. The best validation accuracy achieved was 0.91, while the train accuracy of the best model was 0.9628. This indicates that the model performs well on unseen data; however, the slight difference between the validation and test accuracies suggests that the model may be experiencing some degree of overfitting.

Variations in accuracy could arise from several factors:

- Different random seeds leading to different weight initializations.

- The inherent randomness in the training process, especially with stochastic gradient descent.
- The choice of hyperparameters, which can impact model performance.

In conclusion, the implementation successfully demonstrated the key components of a machine learning pipeline using PyTorch, and the results were consistent with expectations given the chosen architecture and hyperparameters. Further tuning and experimentation could yield even better performance.

5 Figures

We included no relevant figures such as training loss curves and validation accuracy plots to support our findings. We saw that in the check list it was expected figures but we see no reason for them. If we were going to make figures we would maybe make a Confusion matrix, and some visualizations of what picture were wrongly classified.