INF265
Project 1:
Backpropagation and Gradient Descent

Alina Artemiuk & Vegard Sørheim

Spring 2024

Vegard has primarily focused on Section 2, and Alina on Section 3. Throughout the project, we have both collaborated on coding and discussions across these main sections. Given that Section 3 is more extensive than Section 2, Vegard has taken the lead on the report writing.

1 Explanation of our approach and design choices

Section 2 - Backpropagation

The `backpropagation` function for the custom neural network class `MyNet` is designed to compute the gradients of the loss function with respect to the network's weights and biases. There have been a warning when running the second test (`UserWarning: Using a target size (torch.Size([1, 1])) that is different to the input size (torch.Size([1, 10])).[..] return F.mse_loss(input, target, reduction=self.reduction)`), and by consulting Pekka, the mistake is that model should have one-dimensional output. So we created a new test with that adjustment and got no warning.

Vectorized Implementation

The implementation uses vectorization, which is a method of processing data in whole arrays, through PyTorch's operations for matrix and tensor computations for computational efficiency. The vectorized operations are present in the gradients calculation for weights (`model.dL_dw[l]`) and biases (`model.dL_db[l]`), using matrix multiplication (`torch.mm`) and summation across batches of data.

Reverse Layer Iteration

The algorithm iterates over the network layers in reverse order, from output back to input. This reverse iteration aligns with the backpropagation algorithm. This approach ensures that the computed gradients of each layer's parameters are based on the subsequent layer's parameter gradients, adhering to the chain rule.

Gradient Calculations

- **Gradient of Loss w.r.t. Predictions (`dL_dy`):** Initially, the gradient of the loss function with respect to the network's predictions is computed. For a Mean Squared Error (MSE) loss, this gradient is given by `dL_dy` $= -2 \cdot$ (`y_true` $-$ `y_pred`).
- **Gradient of Loss w.r.t. Pre-Activation Outputs (`dL_dz`):** This involves calculating the derivative of the activation function applied to the layer's pre-activation outputs (`model.z[l]`), and element-wise multiplication with `dL_dy` to apply the chain rule. This gives the sensitivity of the loss to changes in the z-values before the activation function is applied.
- **Gradients w.r.t. Weights and Biases:** The gradients with respect to the weights are computed using the outer product between `dL_dz` and the activations from the previous layer (`model.a[l-1]`). For biases, the gradient is the summation of `dL_dz` over its batch dimension, which reflects the biases' contribution to the loss.

Gradient Propagation

After computing the gradients for weights and biases, the algorithm updates `dL_dy` for the next layer in the backward pass. This update ensures that the gradient reflects the earlier layer activations' contribution to the loss. The update involves computing the dot product of `dL_dz` with the weights of the current layer. This is the gradient's backward propagation through the network.

Use of `torch.no_grad()`

The entire backpropagation process is enclosed within a `torch.no_grad()` context. This prevents PyTorch from tracking these operations in its computational graph, because the purpose here is to manually compute the gradients. So, we are bypassing PyTorch's automatic differentiation.

Section 3 - Gradient Descent

3.1 Data Loading and Preprocessing

- The CIFAR-10 dataset was initially loaded and analyzed to understand the distribution of its classes, which is crucial for unbiased model training. The analysis included visualizing the number of images per class and inspecting sample images from each class.
- Normalization was applied to the dataset with a mean of 0.5 and a standard deviation of 0.5 for each RGB channel. We believe that standardization can improve learning process because the images gets transformed into a shared scale without loosing important information.
- The dataset was split into training, validation, and test sets, allocating 80% of the training data for training and 20% for validation, effectively giving 16.7% (2000) to test, 16.7% (2000) to validation, and 66.7% (8000) for training.
- PyTorch's `random_split` function was used to partition the dataset, with the seed previously set to ensure reproducibility.
- Following the above steps, a subset containing only the 'Bird' and 'Plane' classes was extracted to create the CIFAR-2 dataset.

3.2 Class Structure and Initialization

The `MyMLP` class extends `nn.Module`, PyTorch's foundational class for all neural network modules. The `__init__` method initializes the network architecture of four fully connected layers.

- **Input Layer**: The input layer is configured to accept a dimensionality of 3072, corresponding to the flattened size of CIFAR-10 images (32×32 pixels over 3 color channels), facilitating direct processing of raw image data.
- **Hidden Layers**: The network includes three hidden layers with descending units of 512, 128, and 32. This design progresses the feature representation from the input towards the output layer.
- **Output Layer**: The final layer outputs two units for the binary classification. No activation function is applied here because as the cross-entropy loss used later integrates a softmax operation.
- **Activation Functions** ReLU (Rectified Linear Unit) activations are employed across all hidden layers to introduce non-linearity, which allows learning complex patterns.

3.3 Train Function with Optimizer Parameter

1. The function **train** enters a loop to iterate over the specified number of epochs, where each epoch consists of:
   - Initializing an epoch loss accumulator.
   - Iterating over batches of data provided by `train_loader`.
   - For each batch, performing the following steps:
     (a) Moving the input images and labels to the designated computing device.

    (b) Clearing the gradients of all optimized tensors.
    (c) Performing a forward pass through the model.
    (d) Calculating the loss using the specified loss function.
    (e) Executing the backward pass to compute the gradient of the loss function with respect to the model parameters.
    (f) Updating the model parameters based on the gradients.
    (g) Accumulating the loss over all batches to compute the total epoch loss.

- If the current epoch is the first or a multiple of five, printing the timestamp, epoch number, and average training loss for the epoch.

2. For running the function we use 30 epochs, the optimizer `optim.SGD`, the loss function is `nn.CrossEntropyLoss`, and trainloader with 256 batch size and shuffle set to false.

3.4-5-6-7 Train Function with Manual Parameter Updater

We have added together task 4, 5, 6, and 7 into the function **train_manual_update**.
The training process involves the following steps:

1. Initializing momentum buffers for each model parameter to facilitate momentum-based updates.
2. Iterating over the specified number of epochs, where each epoch includes:
   (a) Setting the model to training mode.
   (b) Iterating over batches of data:
   - Performing forward and backward passes to compute gradients.
   - Manually updating each parameter based on its gradient, learning rate, momentum, and weight decay.
   - Zeroing out gradients after each batch to prevent accumulation.

   (c) Computing and printing the average training loss after certain epochs.
3. After each epoch, evaluating the model on the validation set to compute the validation loss.
4. Recording training and validation losses, as well as accuracies for each epoch.

The **manual update rule** applied for each parameter is as follows:

$$p_{\text{new}} = p_{\text{old}} - \text{lr} \cdot (\text{grad} + \text{weight\_decay} \cdot p_{\text{old}} + \text{momentum} \cdot \text{momentum\_buffer}) \tag{1}$$

where,
1. **Gradient Calculation with L2 Regularization**:

```
grad = p.grad
grad += weight_decay * p.data
```

- Here, `grad` initially represents the gradient of the loss function with respect to the parameter `p`.
- The line `grad += weight_decay * p.data` applies L2 regularization by adding `weight_decay` times the parameter value (`p.data`) to the gradient.
2. **Momentum Update**:

```
buf = momentum_buffers[name]
buf.mul_(momentum).add_(grad)
grad = buf
```

- `buf` retrieves the current momentum buffer for the parameter. This buffer stores the "velocity" or accumulated gradient adjustments from previous steps, scaled by the momentum factor.
- `buf.mul_(momentum)` scales the existing buffer by the momentum coefficient (γ), effectively damping the previous velocity according to the momentum term.
- `.add_(grad)` then adds the current adjusted gradient (which now includes the L2 regularization effect) to this scaled buffer, updating the velocity with the latest information.
- The final assignment `grad = buf` sets `grad` to be this updated velocity, which will be used for the parameter update.
3. **Parameter Update**:

```
p.data -= lr * grad
```

- This line updates the parameter `p` by subtracting the product of the learning rate (`lr`) and the adjusted gradient (`grad`, which now represents the momentum-updated gradient including the L2 penalty).

Training Two Instances of MLP - We trained one instance on the `train_manual_update` (which includes the regularization and momentum) and one instance on the `train` including the same weight decay and momentum. Both functions give the same training losses for each epoch and we have verified that they do the same calculations. So, for avoiding unnecessary calculations, we will from now on use `train_manual_update` for training with the hyperparameters.

3.8-9 Training of Six Different Instances of MyMLP

The **hyperparameters** varied across experiments include the learning rate (`lr`), momentum (`mom`), and weight decay. (`decay`). We chose the hyperparameters provided in the text file. The specific values tested were:

- Learning Rate (`lr`): Fixed at 0.01 for all experiments.
- Momentum (`mom`): Varied among 0, 0.9, and 0.8 to assess the impact of accelerating SGD in the direction of consistent gradient descent.
- Weight Decay (`decay`): Varied among 0, 0.01, and 0.001 to evaluate the effect of L2 regularization on model training and overfitting.

The **training procedure** is 30 epochs using manual update function. We are then selecting the best model by the highest validation accuracy.

3.10 Evaluating the Best Model

The code segments perform the following operations:

1. Initialization of the MyMLP model with specified hyperparameters.
2. Training of the model using both built-in SGD optimization and a custom manual update method.
3. Evaluation of the trained model on a test dataset.
4. Visualization of training/validation losses and accuracies, and a confusion matrix to analyze the model's performance.

The model is trained using a manual update function `train_manual_update`, which allows for explicit control over the optimization process. This function takes as inputs the number of epochs, learning rate, model, loss function, training and validation loaders, weight decay, and momentum. It outputs the training and validation losses and accuracies for each epoch.

The function `compute_accuracy_and_get_labels` evaluates the model's accuracy on a given dataset loader and also extracts the true and predicted labels for further analysis.

The performance of the model over training epochs is visualized through plots of losses and accuracies, providing insights into the learning dynamics and the effectiveness of the chosen hyperparameters. `compute_accuracy_and_get_labels`.

2 Questions

a. The PyTorch method that corresponds to the tasks described in section 2 is the automatic differentiation provided by PyTorch's Autograd system. Specifically, manual backpropagation is implemented to compute gradients of the loss function with respect to weights $\frac{\partial L}{\partial w_{i,j}^{[l]}}$ and biases $\frac{\partial L}{\partial b_j^{[l]}}$. However, in practice, PyTorch can automate this process using the `.backward()` method on the loss tensor.

   Here's how PyTorch's Autograd system relates to the tasks in section 2:

1. **Automatic Gradient Calculation**: - PyTorch's `.backward()` method automates the computation of gradients, which is essentially what the manual backpropagation function aims to achieve. When `.backward()` is called on the loss tensor, PyTorch calculates the gradients for all tensors in the computation graph that have the `requires_grad` attribute set to `True`.

2. **Storage of Gradients**: - After calling `.backward()`, the gradients are automatically stored in the `.grad` attribute of the tensors (parameters of the model like weights and biases). This eliminates the need for manually storing gradients in `model.dL_dw` and `model.dL_db`.

3. **Vectorized Implementation**: - PyTorch inherently uses vectorized operations, especially when computing gradients using the Autograd system.

In summary, while the solution manually computes and stores gradients, in PyTorch, the equivalent and more efficient approach would be to:

- Forward pass to compute predictions.
- Compute the loss using a loss function, e.g., `torch.nn.functional.mse_loss` for Mean Squared Error.
- Call `.backward()` on the loss to automatically compute the gradients.
- Access gradients via the `.grad` attribute on model parameters (e.g., `model.parameters()`) for further operations like parameter updates, which can be done manually or using optimizers like `torch.optim.SGD` or `torch.optim.Adam`.

b. Gradient checking is a method used to verify the correctness of gradients computed by the backpropagation algorithm in neural networks. It compares the gradients obtained from backpropagation with numerically approximated gradients, based on the finite differences method.

Numerical Approximation of Gradients
To approximate the gradient of the loss function with respect to a parameter (either a weight $w_{i,j}^{[l]}$ or a bias $b_j^{[l]}$), we change the parameter by a small value ϵ and observe the change in the loss function. The approximation is given by the formula:

$$\frac{\partial L}{\partial w_{i,j}^{[l]}} \approx \frac{L(w_{i,j}^{[l]} + \epsilon) - L(w_{i,j}^{[l]} - \epsilon)}{2\epsilon}$$

Similarly, for biases:

$$\frac{\partial L}{\partial b_j^{[l]}} \approx \frac{L(b_j^{[l]} + \epsilon) - L(b_j^{[l]} - \epsilon)}{2\epsilon}$$

Generalized in vectorized implementation:

$$\frac{d\theta_{\text{approx}}}{d\theta} = \frac{J(\theta_1, \theta_2, \ldots, \theta_i + \epsilon) - J(\theta_1, \theta_2, \ldots, \theta_i - \epsilon)}{2\epsilon}$$

Computing Gradients Using Backpropagation
The gradients $\frac{\partial L}{\partial w_{i,j}^{[l]}}$ and $\frac{\partial L}{\partial b_j^{[l]}}$ are computed using the backpropagation algorithm. These are the gradients that we aim to verify using gradient checking.

Comparing Gradients
The next step is to compare the gradients from backpropagation with the numerically approximated gradients. This comparison is typically made using the relative difference:

$$\text{Relative Difference} = \frac{\|\text{gradient from backprop} - \text{numerical gradient}\|_2}{\|\text{gradient from backprop}\|_2 + \|\text{numerical gradient}\|_2},$$

where 2 is the L2 norm. This is the euclidean distance normalized by the sum of the norm of the vectors.

If we for instance set epsilon to if $1e-7$ and the gradient check returns a value less than $1e-7$, then the backpropagation algorithm is implemented correctly. Otherwise, there is probably a mistake in the implementation which we can be sure of if the value is greater than $1e-3$. These numbers are not consistent across articles, so they should be adjusted according to the specific implementation.

Iterative Parameter Checking

Due to the computational cost, gradient checking is performed on a random subset of parameters. So, gradient checking should only be used for debugging.

c. The manual update process described in Section 3, Question 4, bypasses the use of specific PyTorch optimizer methods for updating trainable parameters:

- Directly zeroing the gradients of each parameter, as opposed to utilizing `optimizer.zero_grad()`. This is achieved by iterating over the model's parameters and calling `model.zero_grad()` so that the gradients from previous iterations do not interfere.
- Manually updating the parameters without the `optimizer.step()` method. This manual update adheres to the gradient descent rule, explicitly adjusting parameters with the formula `param -= learning_rate * param.grad`. These updates are performed within a `torch.no_grad()` context to exclude them from gradient computations.

d. Adding momentum to the gradient descent algorithm increases the speed of convergence by smoothing the trajectory and reducing oscillatory behaviour in the "valleys" of the loss function. This helps to move more faster through flat areas and prevents the algorithm from getting stuck in minor dips, providing a more stable path toward the minimum.

e. Regularization is used in the gradient descent algorithm to avoid overfitting, when the model memorizes the training data, including noise, and fails on new data. It adds a penalty to the cost function, promoting smaller, simpler model weights, therefore improving the model's ability to generalise well to unseen data.

f. As outlined in Subsection 3.8-9, we conducted a hyperparameter search, testing different values for momentum, and weight decay with fixed learning rate.

Best Hyperparameter Configuration

The best combination was found to be a learning rate `lr` of 0.01, momentum `mom` of 0.8, and weight decay `decay` of 0.01, which achieved a validation accuracy of 83.21%. On the unseen data, this configuration resulted in an accuracy of 84.25%.

Confusion Matrix & Classification Report

The confusion matrix reveals a slight tendency of the classifier to mislabel 'Bird' as 'Plane', which is reflected in the classification metrics. Despite this, the precision and recall are well balanced across both classes, with 'Bird' having higher F1-score of 0.85 compared to 0.84 for 'Plane'. The overall accuracy stands at 0.84, indicating that the classifier is fairly reliable in its predictions.

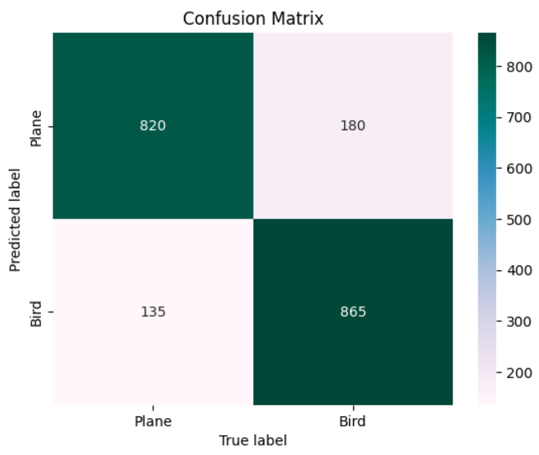

Figure 1: Confusion Matrix.

Class	Precision	Recall	F1-score	Support
Plane	0.86	0.82	0.84	1000
Bird	0.83	0.86	0.85	1000
Accuracy			0.84	2000
Macro avg	0.84	0.84	0.84	2000
Weighted avg	0.84	0.84	0.84	2000

Table 1: Classification Report.

Best Model Performance

The performance plots show the evolution of model loss and accuracy over 30 epochs. The training loss decreases consistently, indicating good learning during training, while the test loss begins to rise after around 20 epochs, suggesting overfitting. The training accuracy shows a steady increase, reflecting the model's improved predictions on the training data. In contrast, the test accuracy plateaus around the 15th epoch, which typically means the model is not generalizing well beyond the training data.

The divergence between training and test performance after a certain number of epochs is a classic sign of overfitting, where the model's ability to predict accurately on the training data is improving, alongside a decrease in performance on new, unseen data. Implementing early stopping could help mitigate this problem.

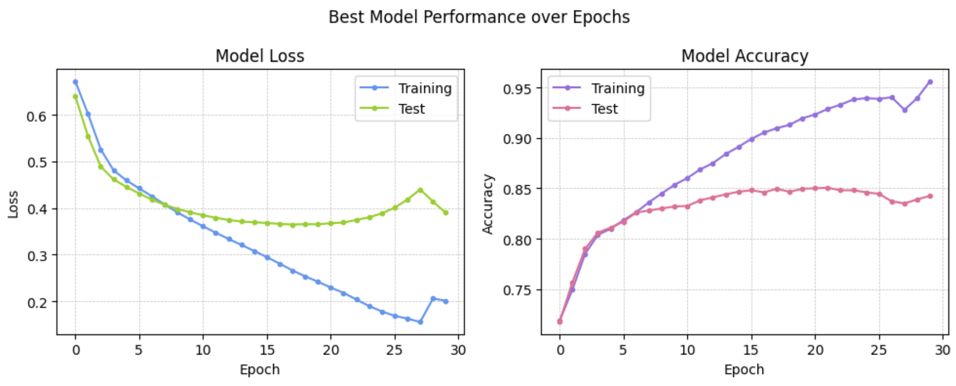

Figure 2: Best Model Performance.

g. **Backpropagation Results**

While working on backpropagation task, we faced a warning due to a dimension mismatch: our network, `MyNet`, had 10-dimensional output for MNIST with 10 classes, but the target labels were in a one-dimensional format, causing issues with calculating the mean squared error (MSE) loss.

Despite passing the tests, we decided to try to solve this problem. We considered three possible solutions: switching to a classification-suited loss function like cross-entropy, one-hot encoding the target labels, or adjusting the network's output layer to match the target labels' dimensionality. Due to the requirement to use MSE loss, we discarded the first option. One-hot encoding did not resolve the issue, so we decided to implement the third solution by modifying the network's output layer to align with the target labels' dimensions. This approach resolved the dimension mismatch, as evidenced by an additional test cell displaying the adjusted output dimension.

Gradient Descent Results

The observed trends in the loss and accuracy plots align with the purposes of momentum and regularization described in d. and e.

The plots show that momentum appears to quicken convergence, however, too much momentum without sufficient regularization can lead to erratic behavior, highlighting the need for balanced approach. Regarding weight decay, the regularization effect is evident where decay is applied, promoting smoother convergence and potentially better generalization.

Hyperparameter tuning has shown that the right combination is the key to increasing stability and generalization in the learning process.

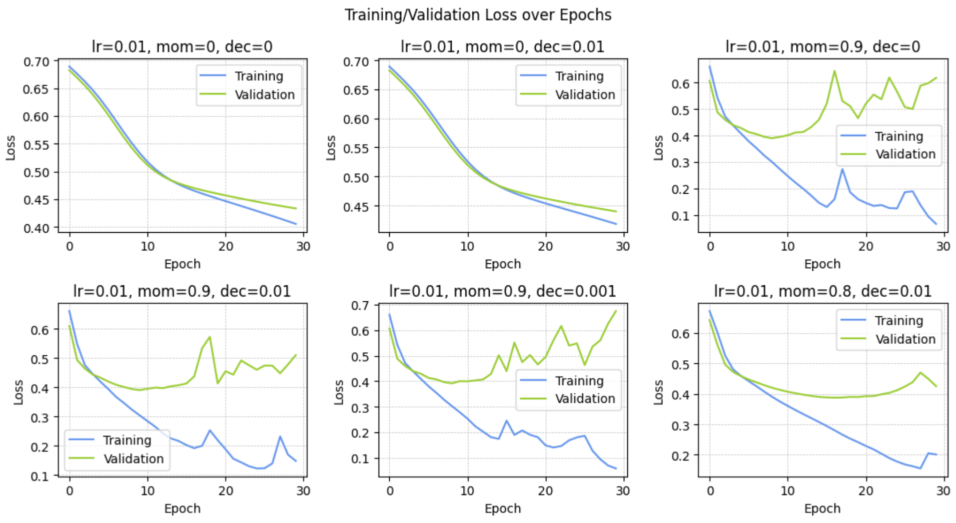

Figure 3: Loss over Epochs.

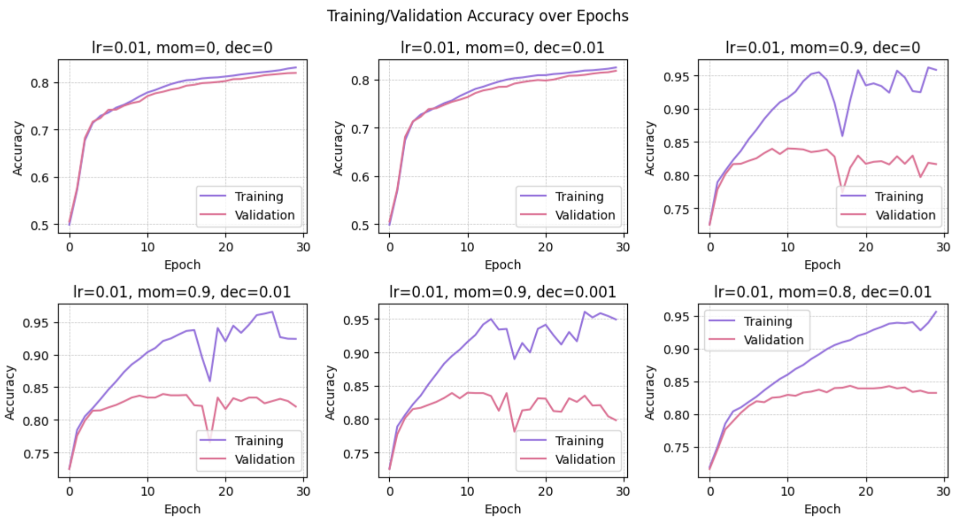

Figure 4: Accuracy over Epochs.