Project 1 Report

1. Approach and design choices.

Backpropagation

In making a backpropagation function for our MyNet model, we focused on calculating the gradients needed to fine-tune the model's parameters. This backpropagation function figures out the gradients of the loss with respect to the outputs, weights, and biases across all the layers of the neural network by using the chain rule from differentiation.

The process starts by calculating the gradient of the loss function with respect to the model's predictions (dL_dy), using the derivative of the sum of squared residuals formula. This is the first step in calculating the gradient for backpropagation. From there, for each layer, starting from the end and going back to the start, the function calculates the derivative of the activation function (da_dz) to get the gradient of the loss with respect to the outputs before the activation (dL_dz).

After this, it's about figuring out the gradients of the loss with respect to the weights (dL_dw) and biases (dL_db) of each layer. This is done using the outputs from the previous layer and the calculated gradients of the loss with respect to the layer outputs. We make sure we can multiply the matrices correctly, especially adjusting for cases where the outputs are one-dimensional.

For layers beyond the first, the function sends the gradient back to the previous layer (dL_da), linking each layer's parameters to the final loss. This backward step through the network layers is the key part of backpropagation, allowing the model to update its parameters in a way that reduces the loss function.

The function runs under torch.no_grad() to stop PyTorch from automating these processes, as we want to manually handle the gradient descent process. This way, the backpropagation function gives us better control over the learning process, separate from PyTorch's automatic calculations and updates."

Gradient descent

Imports and project overview

In my project, I leaned heavily on PyTorch for its dynamic computation graph and extensive neural network modules, which made building and training models feel intuitive. PyTorch's flexibility in designing custom architectures and its straightforward approach to automatic differentiation significantly eased the implementation of gradient descent. For visualizing results, which is crucial for understanding model performance and diagnosing issues, I turned to matplotlib. It allowed me to create clear, informative plots of losses and accuracies over epochs, making it easier to communicate results and insights.

My programming skills were sufficient for this project, thanks to my proactive learning, especially with PyTorch tutorials. This knowledge allowed me to effectively implement gradient descent and tackle the project's challenges without limitations. Given the approach and tools used, my implementation is both efficient and "pythonic," leveraging optimized libraries and following Python's best practices for clear and concise code. My main goal was to Split the problem into smaller functions, which in my opinion worked quite well.

Loading and splitting the dataset

In my project, I utilized the CIFAR10 dataset, applying a standard preprocessing routine to normalize the images and then splitting the data into training, validation, and test sets. Specifically, I reserved 90% of the data for training and the remaining 10% for validation. This split ensures that I have sufficient data for training while still being able to validate the model's performance on unseen data. Additionally, I filtered the dataset to focus on two classes - airplanes and birds - to simplify the classification task. This decision allowed for a more focused study on binary classification, reducing complexity and computational demands. The final dataset sizes, with over 9,000 training samples, nearly 1,000 validation samples, and 2,000 test samples, provided a balanced framework to train, validate, and test my model effectively.

Analyzing CIFAR-10

In analyzing the CIFAR10 dataset, I discovered that each element is a tuple comprising two components: the first is a PyTorch Tensor representing the image, and the second is an integer denoting the label. The distribution is uniform across the ten classes, indicating a well-balanced dataset, which is beneficial for training unbiased models.

I visualized an image from the CIFAR10 dataset after preprocessing, which involved normalization. The printed tensor values, now scaled between -1 and 1, indicate the effect of normalization. Displaying the image validates the preprocessing, showing that despite the transformation, the image retains its recognizable features.

I designed a multi-layer perceptron (MLP) with a specific architecture for a binary classification task on CIFAR2. I utilized PyTorch's functionality to automate most of the process by using its nn.Sequential module. This approach reduces the amount of code by allowing you to define the structure of the neural network and its activation functions succinctly, with PyTorch handling the calculations automatically. The MLP consists of four fully connected layers with ReLU activations for non-linearity, excluding the output layer, as the cross-entropy loss used later includes a softmax function. Analyzing the model, I found it has a total of 1,643,234 parameters, distributed across the layers as per the architecture's design. When testing the model with an input image from the CIFAR2 dataset, it successfully generated an output tensor.

Training MLP

In my training approach for the MLP, I implemented a manual update mechanism for the model's parameters, incorporating momentum and L2 regularization. This method calculates gradients for each batch, updates parameters considering both the current gradients and the accumulated velocity from previous steps and applies weight decay for regularization. The process, iterated over several epochs, aims to minimize the training loss effectively, with progress logged at regular intervals.

Training on different instances of MLP model

Exploring different parameter settings, in gradient_decent_output , I've identified five new configurations aimed at optimizing model training:

lr=0.005, mom=0.9, decay=0.001: Aims for precise updates and better end-stage convergence.

lr=0.02, mom=0.8, decay=0.005: Seeks faster convergence with stability and anti-overfitting measures.

lr=0.01, mom=0.7, decay=0.002: Focuses on finer gradient updates and stronger regularization for improved generalization.

lr=0.015, mom=0.85, decay=0.01: Balances quicker convergence with control over regularization.

lr=0.008, mom=0.95, decay=0.003: Combines smoother convergence with moderate regularization.

Choosing best model

Focusing on the 3 factors high validation accuracy, stable and low validation loss and minimal gap between training and validation accuracy. I chose. The model with the parameters ('lr': 0.01, 'momentum': 0.7, 'decay': 0.002}. Based on the plots you can see that it is the most stable and best performing.

E**valuating model**

After testing it had a training accuracy of 98%, a validation accuracy of 85% and finally a test accuracy of 84.8%. The confusion matrix further validates my choice, illustrating the model's effectiveness in classifying the CIFAR2 dataset. With 846 true positives for class 0 and 850 true positives for class 1, alongside relatively low false positives and negatives, the matrix showcases the model's robust predictive capability.

Reviewing misclassified images highlighted the model's challenges in distinguishing between airplanes and birds, often due to ambiguous shapes or similar colors. These insights point to areas for model improvement, suggesting the need for enhanced feature extraction.

Conclusion

To make this project better, I'd try using more advanced models that are great for pictures, like CNNs, because they're really good at noticing patterns in images. I'd also play around with changing the pictures a bit, like flipping them or adjusting the colors, to help the model learn better. Tweaking settings like learning speed and trying different ways to keep the model from memorizing the training data too much could also help. Using models that have already learned a lot from other pictures and seeing if they can do a better job here is another idea worth exploring. All these steps are about making the model smarter at telling birds from planes by giving it more variety and better tools to learn from.

2. Questions and answers

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

Calculating the derivative of the loss function with respect to the weights and biases of each layer is automatically handled by PyTorch's autograd system, specifically when you call .backward() on the loss. For example, after computing the loss with nn.CrossEntropyLoss(), like loss = nn.CrossEntropyLoss()(y_prediction, y_true), you would call loss.backward() to compute the gradients.

(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 method used to check whether the computed gradient of a function seems correct is called gradient checking. you can gradient check manually by comparing the gradients calculated during your backpropagation with gradients computed using a finite difference approximation.

To check gradients with finite difference approximation, I would slightly alter each weight and bias,

observe the change in the loss, and calculate the gradient as the change in loss divided by the adjustment. I could then compare this with the backpropagation gradients to check their accuracy.

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

Updating all the trainable parameters of a neural network is done by the .step() function. This function is called on the chosen optimizer after calculating gradients with the .backward() function. It utilizes the $\phi_t = \phi_{(t-1)} - \alpha\nabla L(\phi_{(t-1)})$ equation to update the parameters. It is called on the chosen optimizer. Different optimizer will alter this equation in different ways, but SGD uses it as it is.

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

Adding momentum to the gradient descent algorithm helps it move faster and more smoothly towards the minimum point of the loss function. It does this by taking into account the previous direction the algorithm was moving in and using that information to keep going in the same direction with more force. This helps avoid getting stuck in local minima and speeds up convergence to the global minimum, making the optimization process faster and more efficient.

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

Adding regularization to the gradient descent algorithm helps prevent overfitting by penalizing large weights in the model. It does this by adding an extra term to the loss function that encourages the model to keep the weights small. This helps create simpler models that generalize better to new, unseen data, improving the model's performance and reducing the risk of overfitting.

(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 used different parameters for the learning rate, momentum and weight decay for each of the different instances. These are the parameters for each instance:

1. Learning rate = 0.005, momentum = 0.90, weight decay = 0.001
2. Learning rate = 0.020, momentum = 0.80, weight decay = 0.005
3. Learning rate = 0.010, momentum = 0.70, weight decay = 0.002
4. Learning rate = 0.015, momentum = 0.85, weight decay = 0.010
5. Learning rate = 0.008, momentum = 0.95, weight decay = 0.003

The best performing combination of parameters was instance number 3. It gave us an initial training accuracy of 0.95, a validation accuracy of 0.85. After picking it for our final model we trained a new model using the parameters and got a final test accuracy of 84.8%.

(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.

Test accuracy: 84.8%

For our results in general, our accuracies seem stay within the range of 0.8-0.86 over the different instances. The different choices in parameters do not seem to cause a huge difference in the final result.