# Project1 INF265

## 1. An explanation of approach and design choices

Backpropagation

In the backpropagation function we computed $\frac{\partial L}{\partial wi}$ for all weights $w_i$ and $\frac{\partial L}{\partial bi}$ for all biases $b_i$ and stored them in two separate dictionaries. We first computed the output layer gradients separately and then propagate errors backward through all hidden layers in the network. We used a vectorized implementation, avoid unnecessary loops over neurons in each layer.

Gradient decent

Our implementation begins by ensuring reproducibility through a fixed random seed using torch.Generator().manual_seed(SEED). The CIFAR-10 dataset is then loaded and split into training, validation, and test sets. We know from the documentation of the CIFAR-10 dataset and a check on the training data that the data is balanced, therefore accuracy should be a good performance measure for the models.

We use only two classes (car and cat), relabeling them as 0 and 1, and resize the images to 16x16. Before applying normalization, we visualize some sample images, then compute the mean and standard deviation of the training set and use these values to normalize all datasets. Another visualization step follows to confirm the effect of normalization.

For the model, we implement a fully connected multi-layer perceptron (MyMLP) using ReLU activation functions. To train the model, we define two methods: a standard training function using PyTorch's SGD optimizer and a manual implementation that updates model parameters using gradient calculations, momentum, and weight decay. We compare the outputs of these methods to ensure correctness. The dataset is loaded into DataLoaders for efficient batch processing, and we use CrossEntropyLoss as the loss function.

We train five models with different learning rates, momentum, and weight decay values inspired by the “gradient_descent_output.txt”, always resetting the random seed before each initialization, ensuring reproducibility. Both training methods (SGD and manual) are applied to verify that they produce the same results. Training loss, train accuracy, and validation accuracy are stored for all models for comparison. The model with the highest validation accuracy is selected for final evaluation on the test set. This approach compares models and gives insight into how different hyperparameters influence performance.

## 2. Questions

a) Tensor.backward() correspond to the task in section 2. Instead of manually computing gradients using the chain rule, PyTorch provides automatic differentiation.

b) A mathematical method to verify computed gradients is Gradient Checking. It compares the manually computed gradients with a numerical approximation to make sure they match. The method is called the finite difference approximation. For a function f(θ) , the gradient with respect to θ can be estimated using the centered difference formula:

$$\frac{\partial f}{\partial \theta} = \frac{f(\theta + \in) - f(\theta - \in)}{2 \in}$$

Where ∈ is a small epsilon value, f(θ) is the loss function we are optimizing and θ is the parameter, either weight or bias. To implement this in section 2, we would choose a small subset of weights or biases from our model and plug the values into the formula to get the estimated gradient. The new gradient should be compared to our manually computed gradient, with relative error. If the relative error is small, the gradients are likely correct.

c) The method corresponding to manually updating the weights with the equation: $\theta_t = \theta_{t-1} - \alpha \nabla L(\theta_{t-1})$ would be: optimizer.step(). This updates the weights using gradients, weight decay and momentum (if given). All of which are added in the manual version. Because the backward() function in PyTorch add to the .grad attribute for each gradient rather than replacing them we need to zero out the gradients each iteration. The corresponding PyTorch method is optimizer.zero_grad().

d) By adding momentum to the gradient descent algorithm, we achieve a higher converge as momentum accelerates learning by keeping a 'memory' of past gradients. We also achieve smoother updates, as it reduces zigzagging and helps to avoid getting stuck in small local minima.

e) The purpose of regularization is to help discourage overfitting by penalizing complex models. In L2 regularization this is done by adding a penalty to large weights. We used L2 regularization in the train_manual_update method, which adds a penalty to the larger weights by adding their squared sum to the loss function: $Lnew = L + \lambda \sum ||\theta||$

f) The different hyperparameters are learning rate, momentum and weight decay. We trained five separate models of the MyMLP model using different combinations:

| Model | Learning Rate | Momentum | Weight Decay |
|---|---|---|---|
| Model 1 | 0.01 | 0 | 0 |
| Model 2 | 0.01 | 0 | 0.01 |
| Model 3 | 0.01 | 0.85 | 0 |
| Model 4 | 0.01 | 0.85 | 0.01 |
| Model 5 | 0.05 | 0.85 | 0.01 |

For each of these models, we again trained two separate versions using both train() and train_manual_update() to ensure they produce the exact same results as required.

The best model is selected based on the accuracy score on the validation dataset:

| Model | Train accuracy | Validation Accuracy |
|---|---|---|
| Model 1 | 0.897 | 0.885 |
| Model 2 | 0.893 | 0.885 |
| Model 3 | 0.982 | 0.919 |
| Model 4 | 0.983 | 0.921 |
| Model 5 | 0.976 | 0.913 |

(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.

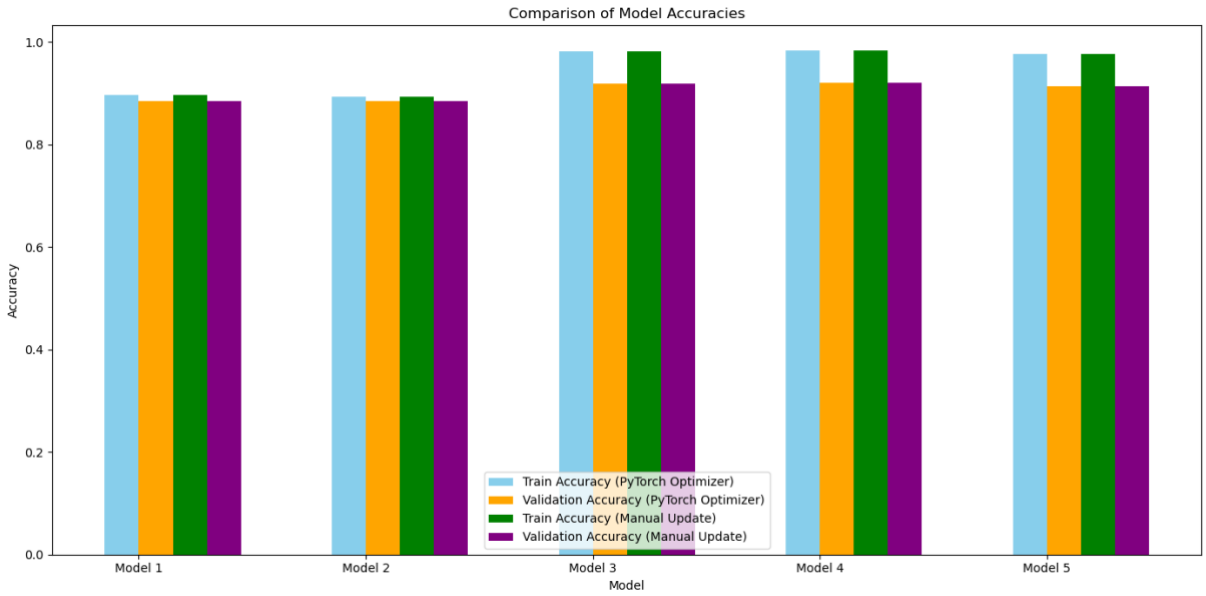


*Figure 1: Training and validation accuracies for all 5 models*

Figure 1 illustrates the training and validation accuracies for our five different models, using both PyTorch SDG optimizer (Blue and orange) and our manual gradient update (green and purple). As observed, our results from both approaches are identical. This is a strong indicator that our manual update function is implemented correctly.

We also notice the impact of momentum on models 3, 4 and 5. These models show a larger gap between training and validation accuracy, compared to models 1 and 2. This suggests slightly overfitting models, as the momentum could accelerate weight updates to aggressively, causing the models to memorize the training data too well. Among these models, model 4 achieves the best balance between training and validation accuracy, whilst also having the highest validation accuracy. This suggests that its combination of momentum 0.85, learning rate 0.01 and weight decay 0.01 provides optimal generalisation. Model 5 has a higher learning rate of 0.05, which means it will converge fast but generalise poorly to training data as seen in its lower validation accuracy and high training accuracy.

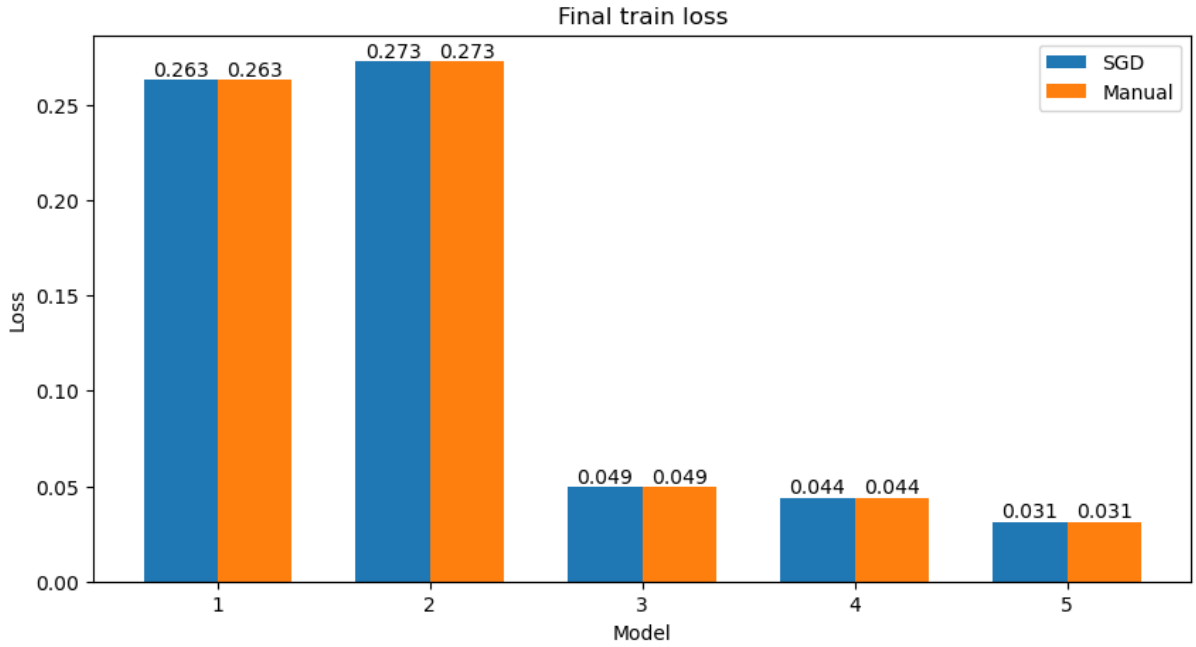


*Figure 2: Final train loss for both SGD and manual implementation for the 5 models.*

For the 5 different models we the tried, the training loss from the final iteration can be seen in figure 2. The two colours represent the SGD implementation and our manual implementation. For the 5 models the bars are the same height and value indicating that the two implementations' calculations are the same.

Interestingly, model 5 achieves the lowest training loss (0,031), but it is not the model with the highest validation accuracy. This may suggest that the model might not generalize well on new data, in other words it might be overfitting.

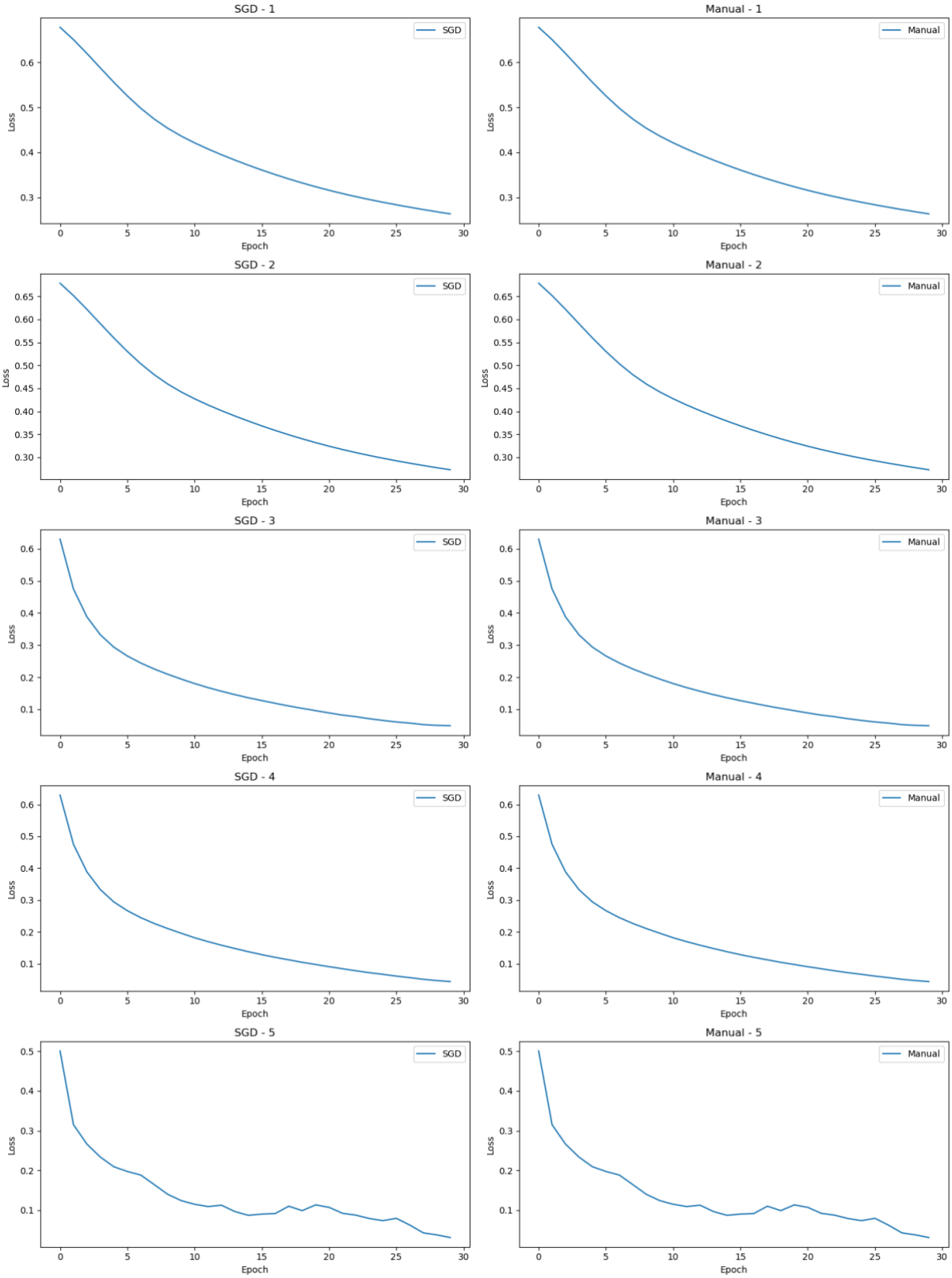


*Figure 3: Shows the training loss over the 30 iterations for both SGD and the manual implementation for the 5 models*

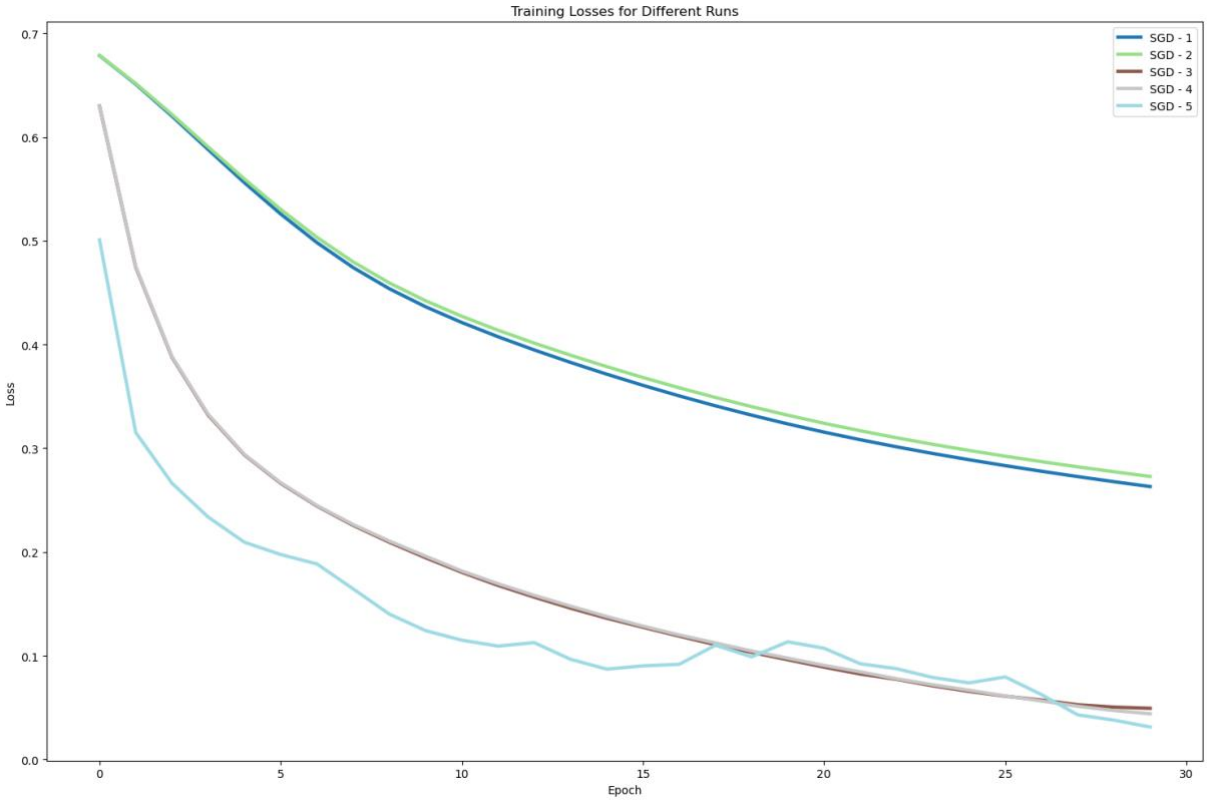


*Figure 4: Training loss over the 30 iterations of all 5 models in the same plot*

Figures 3 and 4 show how the training loss evolves over each iteration, with the final value at iteration 30 represented in Figure 2. Both the SGD and manual implementations are included in Figure 3, and their computations remain identical throughout the training process. In figure 4, we observe that model 5 does not always follow a smooth downward trend; instead, its loss curve has noticeable fluctuations. This is likely due to a high step size, causing the model to overshoot the local minimum and momentarily increase the loss.

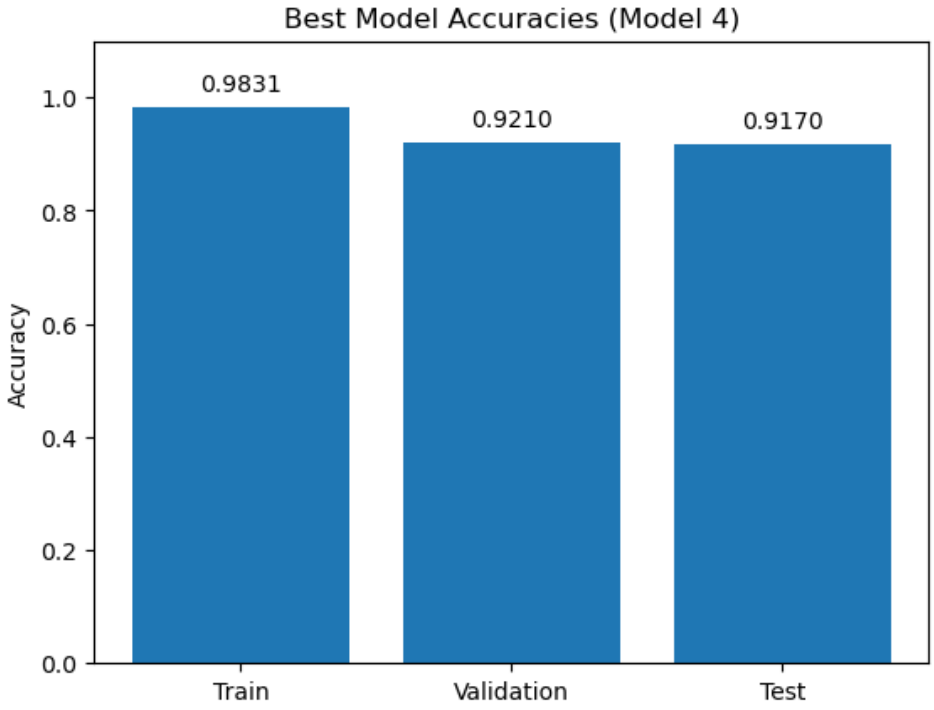


*Figure5: Models 4 (best model) accuracies on train, validation and test data*

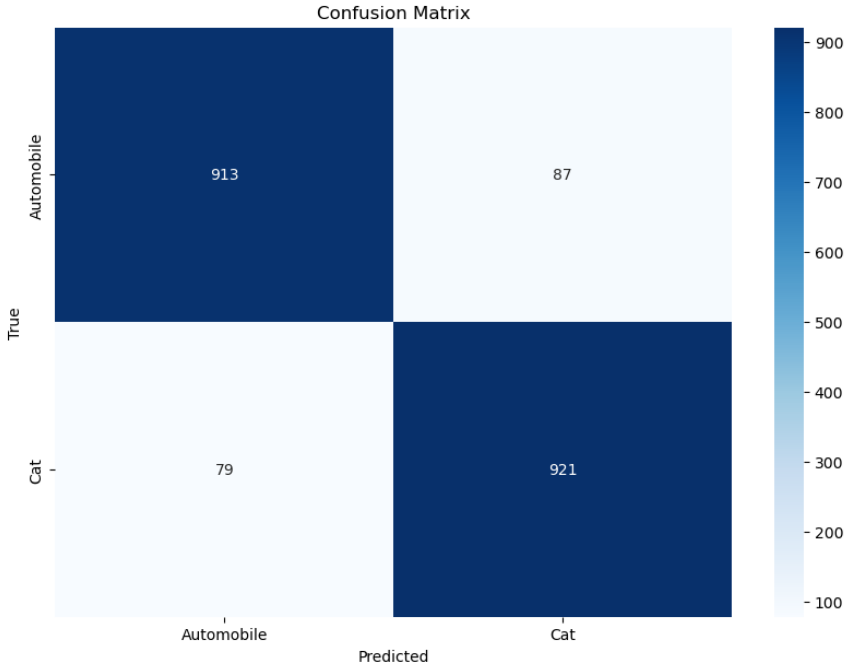


*Figure 6: The correct and wrong predictions of the test-set (model 4)*

To evaluate performance on unseen data, we select the model with the highest validation accuracy. In this case, Model 4 performed best, achieving a validation accuracy of 0.921, with a training accuracy of 0.983 and a test accuracy of 0.917, as seen in Figure 5. The model was trained using a learning rate of 0.01, momentum of 0.85, and a weight decay of 0.01. These parameter choices seem to provide a good balance between stability and generalization, avoiding the instability seen in Model 5 while improving upon Models 1–3.

Figure 6 presents the confusion matrix for the test set predictions of Model 4, illustrating the distribution of correct and incorrect classifications. The matrix indicates that the model correctly classified 913 instances of class 0 (cars) and 921 instances of class 1 (cats), demonstrating strong performance overall. However, there are 87 misclassified cars (incorrectly predicted as cats) and 79 misclassified cats (incorrectly predicted as cars).

The relatively low number of misclassifications suggests that Model 4 has learned to differentiate well between the two classes, with a slight imbalance in errors. The symmetry in misclassifications suggests that the errors are not biased toward one class, meaning the model does not favour predicting one category over the other.