Ensemble Image Classification

ELEC 475

Lab 3

November 19th , 2024

Ryan Zietlow 20347719

Simon John 20348233

Training

The goal of this lab activity is to train three models from the ILSVRC competition from scratch to convergence on the CIFARS-100 dataset. The three models are: AlexNet, VGG16, and

ResNet18. The model architecture can be easily loaded as PyTorch contains the model definitions for all three models. Once loaded the final fully connected layer of the selected model is replaced with one that takes in the same number of parameters but outputs predictions for 100 classes instead of 1000. The final layer needs to be changed because as the name suggests the CIFARS-100 dataset has 100 classes whereas the ImageNet data set that was used for the ILSVRC competition has 1000 classes.

The initial learning rate was set to 0.001 for all models. There is functionality in the code to change the initial learning rate via changing the command used to start training, but the default value is 0.001 so learning rate is omitted from the training commands.

The loss function used is cross entropy loss.

The training of VGG16 required the use of a different optimizer and scheduler than what was used for AlexNet and ResNet18. For AlexNet and ResNet18 the optimizer used is Adam with a learning rate of 0.001 as stated earlier. The scheduler used with Adam is ReduceLROnPlateau, as specified in the code on a plateau the learning rate is decreased by 90% on a plateau with a patience of 3 epochs between plateaus. When training VGG16 the Adam optimizer was not working well so the SGD optimizer was used instead. Like with Adam, SDG used the given learning rate of 0.001, it also used a momentum of 0.9 and a weight decay of 0.0005. When training VGG16 the MultiStepLR scheduler is used instead of ReduceLROnPlateau. MultiStepLR is implemented with milestones of 30, 60, and 90, with a gamma value of 0.1.

In the lab instructions it is specified that the models should be trained to convergence. To determine when to stop training the model, a hyperparameter, patience is created. Patience is the number of epochs the model will continue training after an epoch where the best validation loss is found. When training all models a patience value of 50 is used.

The batch size used for training was determined via trial and error to find a high enough size that most of the dedicated GPU memory is used while avoiding using any shared memory. The use of virtual memory is avoided as it results in much slower epochs. When training AlexNet a batch size of 1400 was used, for VGG16 it was 70, and the batch size was 350 for ResNet18.

For training, a personal computer containing an 8 core AMD 7800x3D, 32 GB of 6000 MHz DDR5 RAM, and a NVIDIA 4070 SUPER was used. The 4070 SUPER has 12 GB of GDDR6X memory, 7168 CUDA Cores, and 568 Tensor Cores.

Table 1 below shows the training time for each model and the number of models trained.

Table 1: Training Times for Various Models

Model	Total Time	Epochs
AlexNet	1:01:06.129	72
VGG16	6:26:15.906	75
ResNet18	4:40:40.514	69

Figures 1, 2, and 3 below show the training plots for all three models

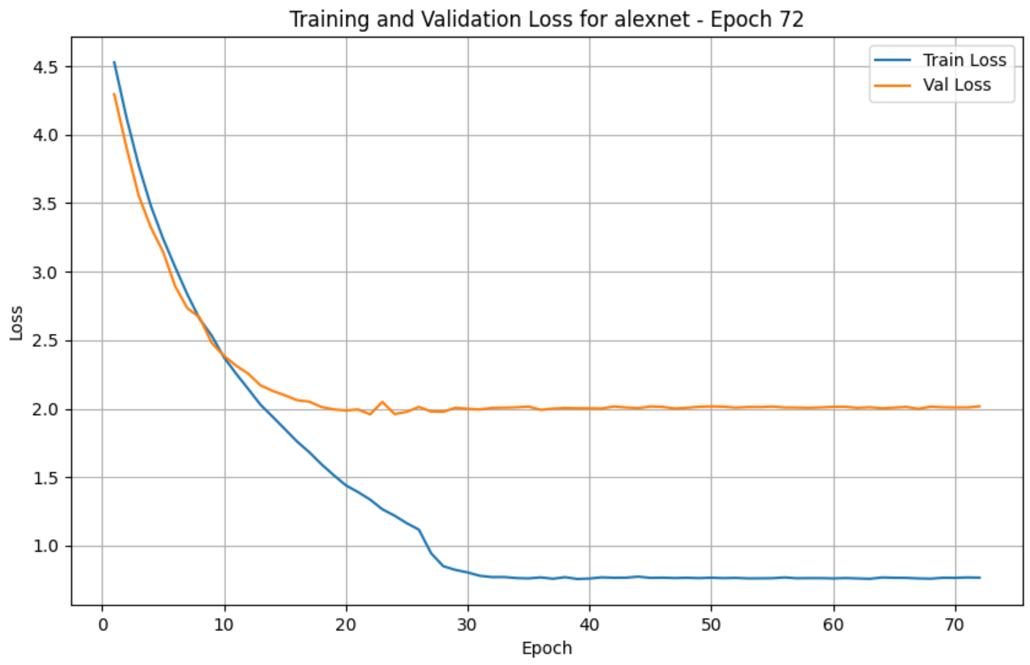

Figure 1: Training Plot for AlexNet

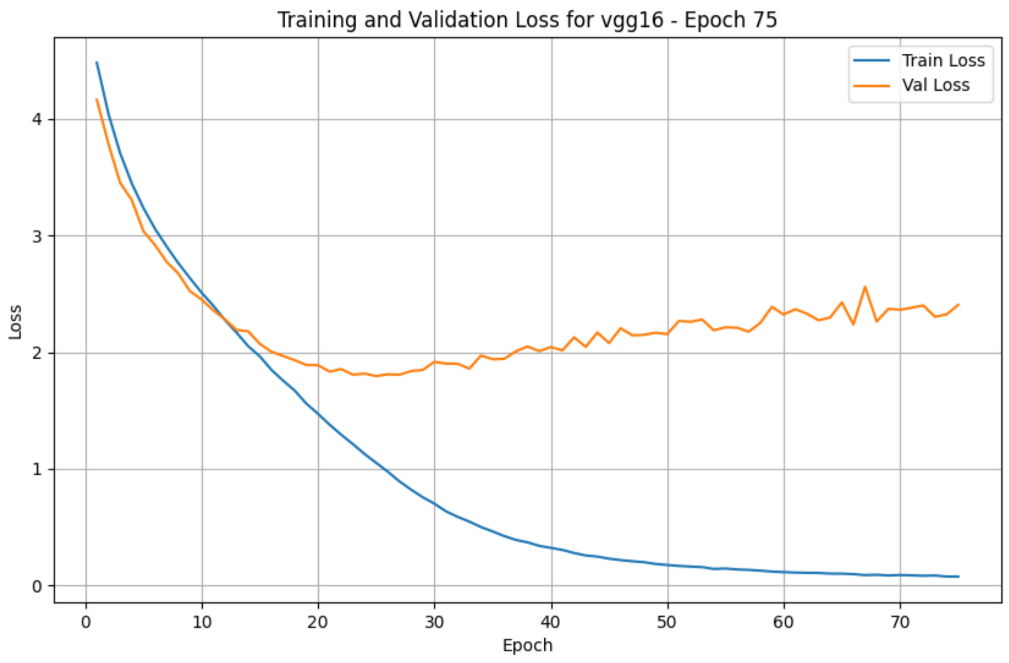

Figure 2: Training Plot for VGG16

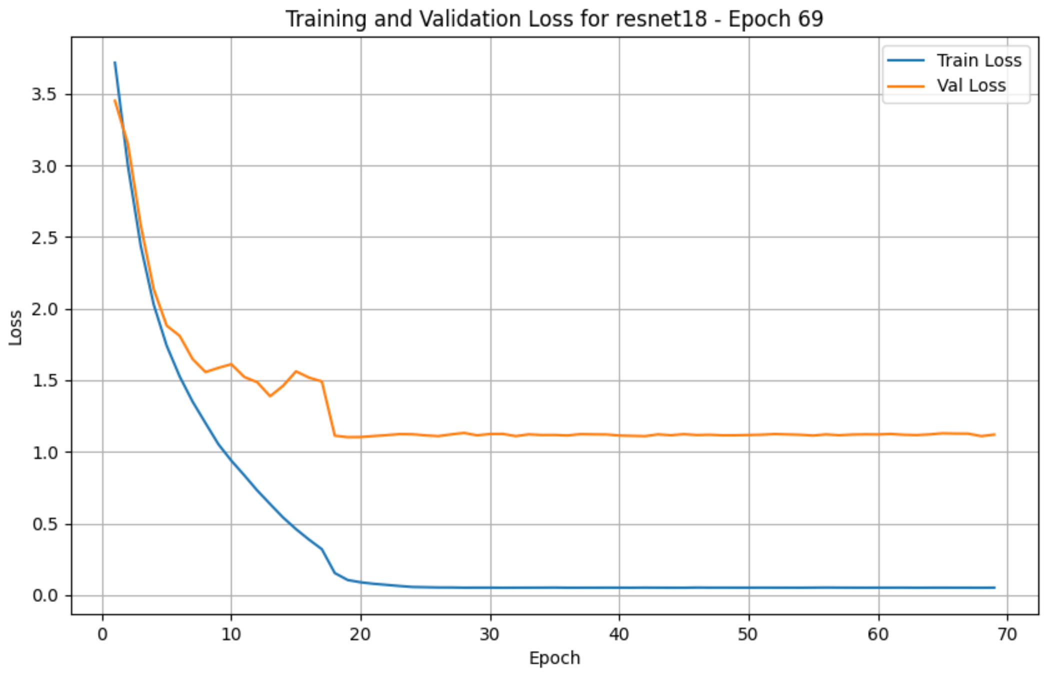

Figure 3: Training Plot for ResNet18

Testing

Table 2: Results of Individual Model Testing

		Error Rate	
Model	**# Epochs**	**Top-5**	**Top-1**
AlexNet	5	47.92%	76.83%
VGG16	5	45.11%	74.21%
ResNet18	5	20.36%	50.91%
AlexNet	72 (Full Convergence)	21.94%	50.01%
VGG16	75 (Full Convergence)	19.54%	46.95%
ResNet18	69 (Full Convergence)	7.96%	28.92%

Ensemble Implementation

Overview

The three different ensemble methods combined predictions from the three neural network architectures: AlexNet, VGG16 and ResNet18. The three models employed for combining the outputs are: max probability, which selects the class with the highest probability from the model outputs, probability averaging, which averages the probability from all models and selects the class with the highest average probability and majority voting, which uses the most frequent prediction from all models.

Implementation of Methods and Strategies

The ensemble model class first defines an initialization method that acts as a constructor for the class. It is called when an instance of the class is created, and its responsibilities include loading the required models and their weights based on the configuration that is provided. To begin, the *device* and *weight_config* are set from the input parameters to define whether the model will run on a CPU or GPU and if the set of weights that should be loaded are *best* or *epoch5*. Next, a dictionary is created and defined to use the model names (‘alexnet’, ‘vgg16’, ‘resnet18’) as keys and to have their corresponding models as the real values. The real values can be accessed via the *select_model* method. Following the dictionary creation, the weight file suffix is determined based off the *weight_config* parameter. The suffix is set to '*_best.pth'* if the *weight_config* is *'best'*, else it is set to *'_epoch5.pth'*. The final part of this method involves loading the model weights. For each model in the dictionary, the weight file path is constructed, the weights are loaded using *model.load_state_dict(),* the model is transferred to the specified device (CPU or GPU) and then the model is set to evaluation mode using *model.eval()*.

The next method in this class is *_get_softmax_outputs*. It retrieves the SoftMax outputs from each model for a given input image. It takes an image as input and passes it through each model and applies the SoftMax function to get the predicted probabilities. The two key details about this method are first of all, the fact that it does not calculate a gradient. Using *with torch.no_grad()* we can ensure that there is no gradient calculation since we will not perform backpropagation during the inference. Second, the SoftMax function is used to convert the raw logits from each model into probabilities that sum to 1 across classes.

The next three methods in the class implement the ensemble strategies. To implement the max probability strategy, the *max_probability* method calls the *_get_softmax_outputs* method to get the SoftMax probability distributions from each model in this ensemble. The output dictionary stores these outputs. Next, we created a tensor where each row was the probability distribution output from one of the models. Then *torch.stack* combined the distributions from each model into a singular tensor of the shape *(num_models, num_classes).* To find the maximum probability for each class across all models, *torch.max* is used and the result is stored in *max_probs* which is a tensor of size (*num_classes)* containing the highest probability for each class. Finally, to retrieve the predicted label, *torch.argmax* finds the index of the class with the highest probability in the *max_probs* tensor which corresponds to the final predicted class label that is returned.

To implement the probability averaging strategy, it begins the same way as the max probability by calling the *_get_softmax_outputs* to get the SoftMax probability distributions from each model for the input image and then combines these distributions from all models into one tensor using *torch.stack*. It then uses *torch.mean* to calculate the average of the probability values across all models along the first dimension which gives us the average probability for each class and produces a tensor shape of *(num_classes)*. Next, it gets the predicted label by using *torch.argmax* to find the index of the highest average probability in the *avg_probs* tensor. It returns this predicted class label.

To implement the majority voting strategy, it begins the same way as the other two by using the *_get_softmax_outputs* to get the SoftMax probability distributions from each model for the input image. Now this is where it differs from the other two, for each models softmax output, the predicted class is found by applying *torch.argmax* along the class dimension which will return us the class with the highest probability for that model. It appends these predictions to the predictions list data structure so that each model has their own list of predictions. Then the predictions for each model are stacked into a tensor of shape *(num_models, batch_size)* where each row corresponds to the predictions of a single model for all images in the batch. Finally, to compute the majority vote, *torch.mode* computes the most frequent value along the first dimension. It returns a named tuple where *.values* contains the predicted class labels.

Data Structures

The main data structure that we used in the ensemble model was a python dictionary. The dictionary called *self.models* was initialized in the *EnsembleModel* class to store the different models (‘alexnet’, ‘vgg16’, ‘resnet’) as keys. In the dictionary, each model's corresponding value is the model itself which is being accessed by the *select_model()* function. This function returns the pre-trained model that is associated with the corresponding key. The dictionary data structure was chosen because once the models are stored, they can be easily accessed by using the corresponding key. Furthermore, the dictionary allowed us to iterate over models during evaluation. In the *evaluate_ensemble()* function, a for loop is used to retrieve each model's predictions. This approach allows us to use the ensemble methods (maximum probability, majority voting, and probability averaging) for each model without having to duplicate code.

Another data structure that we used was a tensor. Each model in the ensemble generates a prediction for an image, which is stored as a tensor representing the predicted probabilities for each class. This tensor is computed using the softmax activiation function that converts raw model outputs to probabilities. For example, in the *EnsembleModel* class, the *_get_softmax_outputs* function collects the SoftMax outputs for each model and stores them in a dictionary of tensors: *outputs[name] = F.softmax(logits, dim=1)*. These SoftMax outputs for each model are in a tensor of shape *[batch_size, num_classes]*, where each element represents the predicted probability of a particular class. The tensors are then used to compute the final predictions for the image depending on which ensemble strategy is chosen.

The last data structure that we used was a list. In the ensemble model we used lists to temporarily store the predictions from each model before performing computations to combine them into a prediction. For example, in the majority voting strategy, we collect the individual predictions from each model for each image in the batch and store these predicted class labels in a list. After collecting the predictions from each model, we can convert the list into a tensor data structure that we explained earlier. The benefit of using a list here is because it's a temporary storage that is more flexible than directly using a tensor since lists allow us to dynamically append predictions.

Ensemble Testing

Table 3: Results of Ensemble Model Testing

Ensemble Method	# Epochs	Top-1 Error Rate
Maximum Probability	5	51.56%
Average Probability	5	50.99%
Majority Voting	5	65.35%
Maximum Probability	72 (Full Convergence)	29.67%
Average Probability	75 (Full Convergence)	30.23%
Majority Voting	69 (Full Convergence)	38.00%

Best Ensemble Method

The best ensemble method at 5-Epochs was the average probability method which had a Top-1 Error Rate of 50.99%, slightly better than maximum probability which was at 51.56%. This result suggests that average probability can smooth out individual model errors at the early stages of training since averaging the models' outputs reduces the noise introduced by less converged models. This leads to a more stable and reliable prediction compared to one that relies on the highest confidence score of one model in particular.

The best ensemble method at full convergence is the maximum probability which had a Top-1 Error Rate of 29.67% compared to average probability which was at 30.32%. This result indicates that as models become more reliable after full convergence, the maximum probability strategy benefits the most from selecting the most confident prediction from each model. This is because when models are trained, their predictions are more accurate, and when picking the one with the highest confidence we will more likely than not receive a better result.

Performance at 5-Epoch vs. Fully Converged

There was a very noticeable difference between the 5-Epoch ensemble models and the fully converged ensemble models.

At 5 epochs, the models are in what's considered early training stages so their individual predictions may not be reliable. This means the models are undertrained, so their weights are not fully optimized, and this will affect their individual predictions which directly impacts its performance. At 5 epochs, maximum probability underperforms because it is making unreliable predictions and selecting the highest probability from one model isn’t effective when the model's confidence is low. Average probability performs better because averaging the probability from individual models helps reduce noise from model errors adding a bit more stability, but the model is still underperforming. Majority voting performs very poorly at 65.35% Top-1 Error Rate because the models are making differnet predictions leading to incorrect ensemble decisions when they cannot reach a consensus.

At full convergence, the models perform much better as they can make more accurate predictions. Maximum probability performs the best since the models now produce accurate predictions so selecting the most confident prediction from a single model will result in a reliable prediction. Average probability still performs well but is slightly less effective since averaging predictions still smooths out errors, but it does not leverage the increased accuracy of a fully trained model. Majority voting is still the worst strategy, even with full convergence. This is since majority voting still relies on all models agreeing which is ineffective if the models are not always in consensus even when fully trained.