REPORT - DEEP LEARNING 2023

Matthew Utti
University of Oulu
Pentti Kaiteran Katu 1, 90570, Oulu
matthew.utti@student.oulu.fi

Abstract

This study investigates the use of remote sensing applications, focusing on the EuroSAT dataset. PyTorch would serve as a tool in conducting a comparative analysis of popular deep learning architectures such as ResNet18, VGG11_BN, Vit_B_16, and Shufflenet_v2_x0_5. Afterward, an evaluation of model performance would be conducted on a small subset of EuroSAT images along with a thorough investigation using different optimization strategies, including Adam and SGD, to assess their impact on training and validation accuracy. This methodology would involve pre-training on miniImageNet, fine-tuning on EuroSAT, and iterative testing with limited image samples. The extract would also feature various strategies for improving model performance, such as data augmentation, transfer learning variations, hyperparameter tuning, ensemble methods, and regularization techniques.

1. Introduction

First, this study would discuss the broad scheme of deep learning as well as its significance and purpose. There would also be an inference into reasons why deep learning falls short in certain scenarios. Typically, transfer learning would take center stage in addressing scenarios with limited sample sizes.
This study embarks on a journey into the intricate world of deep learning, employing the versatile PyTorch framework to unravel the potential of three distinctive models—ResNet18, VGG16, and a custom Vision Transformer—when confronted with the EuroSAT dataset. It also represents a narrative that navigates through the challenging terrain of limited data, probing the adaptability and resilience of these models First, the models would learn universal features on miniImageNet, and then undergo some fine-tuning on EuroSAT with a select group of images.

1.1 What is Deep Learning?

Deep learning is a powerful subset of machine learning, which is in turn a branch of artificial intelligence. It is a technique used to create predictive models or make decisions based on data, without requiring explicit programming for a specific task. Deep learning models are built using artificial neural networks that are capable of learning and representing complex patterns in the data. These neural networks have multiple layers, which is why they are referred to as "deep". This architecture allows them to process and analyze vast amounts of data, enabling them to make more accurate predictions and decisions.

1.2 Deep learning encounters scenarios where it falls short for the following reasons:

Data Constraint: To achieve accurate results, deep learning models often need a significant amount of labeled data. However, in cases where data is limited, these models may find it challenging to learn meaningful patterns and end up overfitting the available data. Therefore, it is essential to identify ways to improve the learning process and optimize the use of available data to enhance the models' performance.

Transferability: While basic deep learning models may not be suitable for every domain or task, they can still be a useful starting point. With some fine-tuning and adaptation, they can be made to perform well in new situations. However, this may require some additional training or custom modifications specific to the new domain.
Computational expenses: To overcome the challenge of limited computational power in certain settings, there are

several strategies that researchers and developers can consider. One option is to optimize the model architecture and hyperparameters to reduce the computational demands without sacrificing performance. Another approach is to use pre-trained models or transfer learning techniques that can leverage existing knowledge and reduce the need for extensive training. Additionally, cloud-based services and distributed computing can provide access to high-performance computing resources without the need for local hardware. By exploring these options and tailoring them to their specific needs, researchers and developers can overcome the limitations of computational power and continue to make progress in the field of deep learning.

Complex relationship within data: Deep learning techniques have come a long way in capturing complex data relationships. However, it is still a challenging task to accurately represent these intricate relationships, especially with basic deep learning methods. The good news is that advanced deep learning techniques can now identify patterns, make predictions, and provide insights that can help businesses make informed decisions. It is crucial to use the right deep learning techniques for handling these complex data relationships effectively. By doing so, we can ensure that businesses make optimal decisions based on accurate insights and avoid any incorrect conclusions.

1.3 What is Transfer Learning?

Transfer learning is a highly effective machine learning technique that helps to save considerable time and resources that would otherwise be required to create a new model from scratch. By leveraging the knowledge gained from pre-training on a related task, transfer learning enables the model to adapt to new tasks with less data and training time. This approach has proven to be very useful for various applications like computer vision, natural language processing, and speech recognition, among others.

1.4 Addressing scenarios of limited sample sizes with transfer learning

Transfer learning is a valuable approach for addressing scenarios with limited sample size for the following reasons:

Transfer of knowledge: Transfer learning is a constructive and time-saving machine learning technique that allows one to reuse pre-existing knowledge learned by a pre-trained model on a related task. This approach enables you to build on the knowledge that has already been gained, saving valuable time and resources. By leveraging the learned features and parameters of the pre-trained model, one could significantly improve the accuracy and efficiency of a model. Transfer learning is particularly useful in scenarios where the amount of data available for the new task is limited, as the pre-trained model provides a solid foundation on which to build.

Model Fine-tuning: Fine-tuning a pre-existing machine learning model is a powerful technique that allows one to create more accurate and effective models while saving time and resources. Rather than starting from scratch, there’s an opportunity to begin with a pre-trained model that has already learned from a larger dataset. Hence, the parameters of the pre-trained model are adjusted to fit the specifics of your task, allowing it to retain its general knowledge while also being customized to meet your needs. This approach is particularly useful when working with limited datasets, as it allows you to work more efficiently and effectively. Fine-tuning a pre-existing model means one could produce models that are characteristic of more accurate and effective outcomes.

Enhancement of model generalization: Transfer learning has been proven to be a promising approach for improving the accuracy of machine learning models. By leveraging the knowledge gained from pre-trained models, transfer learning enables the model to generalize better to new and unseen data, even when the dataset is small. This can lead to more accurate and effective predictions, which can be a valuable asset in a wide range of applications. By using transfer learning, developers can save time and resources while still achieving high-quality results.

1.5 Content of the paper and methods applied.

The goal of this project is to develop a deeper and more nuanced understanding of the performance of various deep learning models on the EuroSAT dataset, in the realm of remote sensing applications . By comparing and analyzing the effectiveness of ResNet18, VGG, and a custom Vision Transformer, we hope to identify areas for improvement and optimization in each model. We will also explore different optimization strategies to determine which ones are most effective in enhancing the accuracy of these models during both training and validation phases.
This made use of convolutional neural network for image classification using transfer learning, as well as key methods and tools such as ResNet18, which represents the architecture within the ResNet (Residual Networks) family of deep neural networks. ResNet18 is often employed for image classification tasks. Its deep architecture with residual connections helps in learning complex hierarchical features from images, making it

suitable for tasks where the model needs to recognize and classify objects within images.
Its potent tendency towards image classification tasks ensures that it’s frequently used as a pre-trained model for transfer learning. ResNet18, like other ResNet architectures, is capable of learning rich and hierarchical features from images.

2 Approach

2.1 First step

Incorporated in the mix are dependencies such as matplotlib, numpy, torchVision as well as a few sub-dependencies from torchVision such as datasets, models, transforms and read_image.
The matplotlib suffices to serve as a visualization tool while numpy dealt in numerical operations and data processing as the model build got underway. The datasets from the torchVision provides access to popular datasets for computer vision tasks. The models encompassed pre-trained models for image classification and typically served as a starting point for the model build. The transform would serve the purpose of data augmentation and processing of the images before feeding them onto the neural network. The torch.optim serves to provide the required optimization algorithm and the Lr_schedular adjusts the learning rate. The stepLR is key to reducing the learning rate after a pre-defined number of epochs. For a start the learning rate which determines the steps taken during optimization was defined with a value of 0.001 while the weight decay which serves as a regularization technique received a value of 0.0005 – This along with 64 various classes were subjected into 15 iterations or epochs.
Next a dictionary was created which incorporated image data processing in the context of training tuning and testing of the model. This suffices to resize the images to possess a height and width of 224pixels, at brightness, contrast and saturation levels of 0.4 each. The images would be randomly flipped horizontally and vertically and Dussian blur would come in handy with random kernel sizes and sigma values. The images would be randomly rotated at angles between 0 and 30 degrees. There would also be an adjustment of sharpness and auto contrast. The images would be randomly equalized and converted into a pytorch tensor.
Then, a function defined to create data loaders for training, validation and test sets. The aforementioned dictionary came in handy here in the transformation of data for different sets. The function calculates the number of samples for each split, and used the torch.utils.data.random_split to split the original dataset into training, validation and testing sets., while specifying the batch size, shuffle, and the number of workers for parallel data processing. The function returns a dictionary containing the data loaders for each set representing the class labels and subsequent dictionary containing the sizes of each split.

2.2 Second step

Next, the device was set-up for computation and preparation of data loaders for training, validation and testing. It checks if a GPU (CUDA) is available using torch.cuda.is_available(). If a GPU is available, it sets the device to "cuda:0"; otherwise, it uses the CPU ("cpu"). This allows the code to dynamically adapt to the available hardware. It ensures that the code can seamlessly switch between CPU and GPU based on the availability of CUDA-compatible devices. The use of separate data loaders for training, validation, and testing allows for efficient handling of data during different phases of model development and evaluation. Then came an extraction of the individual data loaders for training, validation, and testing from the above dictionary. These loaders will be used during different phases of the deep learning pipeline such as training, validation, and testing.
There were some random selection of a specified number of images from a dataset to be displayed in a grid using torchvision and a custom read_image function. It iterates through the randomly selected samples and extracts the image file paths. The images are then appended to a list. It sets the variable x to the desired number of random samples and then uses random.sample to randomly select x samples from the dataset. It uses torchvision.utils.make_grid to create a grid of images from the above list. Then the imshow function displays the image grid.
Next, there was a function definition to evaluate the model that checks if CUDA (GPU) is available and moves the model to the GPU if possible, sets the model to evaluation mode using net.eval(), iterates through the data loader and evaluates the model's accuracy on the validation or test set and returns the accuracy. The succeeding function sufficed to train the model – This checks if CUDA (GPU) is available and moves the model to the GPU if possible, creates a temporary directory to save training checkpoints,

initializes variables to track the best model's accuracy and log losses during training, iterates through the specified number of epochs. For each epoch, it iterates through the training and validation phases, computes and prints the loss and accuracy for each phase. It Saves the model with the best validation, displays a plot of training and validation losses, loads the best model weights and returns the trained model. Then comes a definition of sets up the CrossEntropyLoss function a commonly used loss function for training classification models in PyTorch.

2.3 More information about the approach

The performance of the model was evaluated in concurrent steps of moving the model to the GPU, setting the model to evaluation mode, iterating through the batches in the dataset, calculating the model predictions and accumulating correct predictions to calculate the return accuracy.
Subsequently, hyperparameters were defined to engineer how quickly the model would, regularization strength and how optimization progresses over epochs.
Batch size: defines the number of training samples utilized in one iteration.
Learning rate: determines the step size at each iteration while moving around the minimum value of the loss function.
Weight decay: represents a regularization term that penalizes large weights towards preventing overfitting.
Epochs: is one complete iteration through the entire training dataset.
Momentum: improves the convergence speed and helps overcome local minima. It's multiplies the gradient of the preceding step to determine to determine the next step's direction.
These took the following forms: batchsize = 256, learning rate = 0.0001, weight decay = 0.0001, epochs = 10, momentum = 0.09, step size = 2.
Next, the MiniImageNet dataset was prepared for training the neural network through combining a list of image transformations, including resizing, color jitter, horizontal flip, Gussian blur, random crop, conversion onto pytorch tensor and normalization.
The learning rate scheduler using StepLR adjusts the learning rate based on the number of epochs. It is set to decay the learning rate by a factor of gamma after every step_size epochs.
Next is a function which visualizes the predictions of a given model on a subset of images from a validation dataset. It also defines a helper function imshow for displaying images. It takes a PyTorch model, a dictionary of data loaders , the number of images to visualize, and the class names. It sets the model to evaluation mode using model.eval() and stores its original training mode. It iterates through the validation data loader and makes predictions using the model. For each batch of images, it displays the actual and predicted labels along with the corresponding images using the imshow function. The loop stops when the specified number of image is reached.Finally, it restores the model's original training mode.

3 Experiments

The fine-tuning of the model involved the creation of an instance for training and testing with specified parameters for the EusoSAT sataset. It also provides flexibility in selecting a subset of classes, shuffling images and specifying the number of images per class.
The "getDataset" function sufficed to obtain a copy with specific parameters to group the images by their respective classes for better organization and to train your model more effectively. Then, the images were selected to create a training set and a test set. This could help the model to generalize well on unseen data. The size of the sets can be adjusted based on the size of the original dataset and the task at hand.
After creating the sets, transformation functions were defined for both the training and testing stages. These functions could be used to augment the dataset and improve the model's performance using a transformation technique known as normalization.
The model performance was optimized by passing the custom Dataset instances to the DataLoader constructor, along with other parameters such as batch size, shuffle, and the number of workers, you can load data in batches and pass it to the model for training or inference. This will help to optimize the training process and improve the model's performance.
The train_transforms are applied during the training phase to augment the dataset and improve the model's ability to generalize. The following transformations are applied:
Resize (224): All images are resized to a consistent size of 224x224 pixels. This standardization ensures that the input size to the model is consistent.
Random Horizontal Flip: Images are randomly flipped horizontally. This transformation introduces diversity in the training data, enabling the model to learn from different orientations of the same object.
Random Vertical Flip: Similar to horizontal flip, random vertical flipping is applied to further diversify the dataset. This transformation helps the model become robust to variations in vertical orientations.
ToTensor(): Converts the image into a PyTorch tensor. This is a fundamental step, as deep learning models, especially convolutional neural networks (CNNs), operate on tensors.
Normalize((0.4, 0.4, 0.5), (0.6, 0.6, 0.6)): Normalizes the pixel values of the image. Normalization is crucial for stabilizing training, as it ensures that the input features

have a similar scale. The specified mean and standard deviation values are used for normalization.
The test_transforms are applied during the testing phase to preprocess the images for model evaluation. These transformations aim to ensure that the testing conditions are consistent with the training conditions:
Resize (224): Similar to the training phase, images are resized to 224x224 pixels for consistency with the model's input expectations.
ToTensor(): Converts the image into a PyTorch tensor. This step is necessary for the model to process the image during inference.
Normalize((0.4, 0.4, 0.5), (0.6, 0.6, 0.6)): Normalizes the pixel values using the same mean and standard deviation values as used during training. This ensures that the testing data is processed in a manner consistent with the training data.
The eval function is designed to assess the performance of the neural network on a given dataset. Key characteristics of this function include: net(the neural network being evaluated), data_loader(The DataLoader providing batches of data for evaluation) and the criterion(the loss function used for calculating the performance metric).
The function sets the network to evaluation mode using net.eval(). It iterates through batches of data from the provided data_loader and for each batch, the model makes predictions (outs) and computes the loss against ground truth labels (labels) using the specified loss function. The correct predictions and total number of images are accumulated and the function prints real-time progress, indicating the current batch, correct predictions, and the total number of processed images - It returns the accuracy and average loss over the entire dataset afterwards.
The train function is responsible for training the neural network over multiple epochs. Key features of this function include: net(the neural network to be trained), train_loader(the DataLoader providing training data), test_loader(the DataLoader for testing the model during training), num_epochs(the number of training epochs), learning_rate(the learning rate for the optimizer) and weight_decay(for regularization).
The function Initializes the loss function (nn.CrossEntropyLoss()), optimizer (Adam), and learning rate scheduler, then sets the network to training mode using net.train(). It Iterates through epochs and training batches, calculates the loss and performs backpropagation to update model parameters before printing real-time training progress, including epoch, batch, and current loss. Then it Invokes the eval function to evaluate the model on the test set, providing insights into the model's generalization and adjusts the learning rate using the scheduler.
The train_transforms are applied during the training phase to augment the dataset and improve the model's ability to generalize. The following transformations are applied:
Resize (224): All images are resized to a consistent size of 224x224 pixels. This standardization ensures that the input size to the model is consistent.
Random Horizontal Flip: Images are randomly flipped horizontally. This transformation introduces diversity in the training data, enabling the model to learn from different orientations of the same object.
Random Vertical Flip: Similar to horizontal flip, random vertical flipping is applied to further diversify the dataset. This transformation helps the model become robust to variations in vertical orientations.
ToTensor(): Converts the image into a PyTorch tensor. This is a fundamental step, as deep learning models, especially convolutional neural networks (CNNs), operate on tensors.
Normalize((0.4, 0.4, 0.5), (0.6, 0.6, 0.6)): Normalizes the pixel values of the image. Normalization is crucial for stabilizing training, as it ensures that the input features have a similar scale. The specified mean and standard deviation values are used for normalization.
The test_transforms are applied during the testing phase to preprocess the images for model evaluation. These transformations aim to ensure that the testing conditions are consistent with the training conditions:
Resize (224): Similar to the training phase, images are resized to 224x224 pixels for consistency with the model's input expectations.
ToTensor(): Converts the image into a PyTorch tensor. This step is necessary for the model to process the image during inference.
First, it normalizes the pixel values using the same mean and standard deviation values as used during training. This ensures that the testing data is processed in a manner consistent with the training data.
Then came a careful analysis of the curves which showed a balance between model complexity and generalization, leading to a well-performing Vision Transformer model.

4. Conclusion

The hyperparameter tuning and training iterations report provided insights into the key settings influencing the training process. The loss plot analysis offered a visual understanding of the model's learning dynamics and generalization while the iterative training approach allowed for robustness assessment and optimization of hyperparameters for improved performance. The loss plot analysis provides a visual representation of the model's learning dynamics and generalization capabilities – therefore, achieving a well-performing model requires a balanced consideration of hyperparameters and continuous monitoring of training progress. Hence the insights gained from these reports serve as a foundation for future model refinement and optimization.

5 References

https://pytorch.org/docs/stable/index.html

Hanwen Liang, Qiong Zhang, Peng Dai1 and Juwei Lu1 -
Boosting the Generalization Capability in Cross-Domain
Few-shot Learning via
Noise-enhanced Supervised Autoencoder