LEARNING FROM LIMITED DATA WITH RESOURCE CONSTRAINTS

Hussein Abdallah Trong-Tuan Tran Manh-Quoc-Dat Le

1. Introduction

Learning with limited data are one of the interesting aspects of Deep Learning (DL) in Machine Learning (ML) field in general. Through this project we have a chance to experience this challenge in two different contexts.

1.1. Project description:

Challenge 1: Few-sample learning on the CIFAR-10 dataset. We will consider a dataset with only 100 samples. For testing a larger dataset will be used (e.g. the CIFAR-10 test set). The task is to train on 100 randomly selected (but class balanced) samples from the CIFAR-10 training set.

Final evaluation: In the final evaluation your model will be run on 5 random instances of this problem. Evaluation will occur both on the CIFAR-10 test set and other data which will not be revealed at the moment. There is also a CodaLab competition in this challenge 1 and the CodaLab environment only supports CPU, not GPU.

Time constraint for challenge 1 is 2 hour model running time in GPU.

Challenge 2: Consider the same exercise with challenge 1 but now with the ability to use an external data or models not trained on CIFAR-10 from the pytorch model repository or to train your own model on external data (you may not train a model on CIFAR-10 dataset or any derivative using these images. Teams may consider for example various forms of fine-tuning, meta-learning, semi-supervised learning with an external dataset and other related ideas).

For both challenges, during development phase, the team should train and evaluation on the train set, while the final result of training and testing on CIFAR-10 Test set need to be reported with the average and standard deviation. This should be done on a minimum of 3 splits. Each split should use 100 training samples and a separate 2000 samples for testing as done in the test bed when sampling from CIFAR-10 Train set.

1.2. Goals

- For challenge 1, due to the present of the competition in CodaLab [1], our goal is not just simply reaching max test accuracy, but there is more things to be taken into account. Due to limited sample size (100 samples without using pre-trained model) and randomness, variation of model performance between each running instance plays a big role. From early development phase the team observed that it's not so difficult to increase model accuracy when it has not reached its full potential, but the problem is there is too much variances on the final test accuracy between runs within the same code and hyper-parameters, as well as there can be huge differences in performance between running in the training set from our test-bed v.s. running in CodaLab test set, which will be discussed further in the implementation and result section 3.1.5. Another factor to consider is execution speed: since the CodaLab environment for the competition can support running with CPU but not GPU and the maximum execution time allowed is 2 hours, the submitted model on CodaLab should be able to running in CPU within the limited time frame. Hence, our goal for challenge 1 is to achieve the best test accuracy with acceptable execution time (not much longer than the provided baseline CNN model), and most importantly, **the model should have maximized regularization, or minimized variation across running instances for both local and CodaLab execution**.

- For challenge 2, many pre-trained models are found that you can start with to classify your few samples problem. Many of them are trained with ImageNet data-set [9] that has more than 20,000 categories including their annotations. The problem here is which model is more suited to your problem and how deep that model is to generalize well using your limited data. In addition to model selection, many questions will appear that need more investigations such as: how to find similar classes data-set to train your model with? What data augmentation and fine tuning well suite your data? What are optimization techniques you need to look at it carefully to achieve higher accuracy? And finally, how much resource and time your model will need during training and inference time?. Thus, our goal for challenge 2 is to show how a very deep convolution neural network can be used to learn from small data-set like CIFAR-10 with few samples with

availability of using external data to achieve higher accuracy with low variance.

1.3. CIFAR-10 Dataset

CIFAR-10 dataset [22] is an abbreviation of Canadian Institute For Advanced Research. It is one of the most frequently used datasets for training machine learning (ML) models and computer vision algorithms. According to Wikipedia [6], CIFAR-10 dataset consists of 60,000 32x32 color images in 10 different classes, which are airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. There are 6,000 images per class. From the total of 60,000 images in the dataset, CIFAR-10 is split into training and testing sets, which contain 50,000 and 10,000 images, respectively. As the name already implied, training dataset is used for researchers to design and implement their algorithms/approaches and testing set is used for evaluating their solutions/models.

CIFAR-10 image set is mainly used to teach a computer how to recognize or classify objects through photos. Due to low resolution of images (32x32) in the dataset, CIFAR-10 is a reasonable choice for data scientists/engineers could come up with different algorithms or models rapidly to conclude which is the best. There are numerous models based on Convolutional Neural Network (CNN) [23] having high accuracies at classifying images in the CIFAR-10 dataset.

1.4. CINIC-10 Dataset

CINIC-10 is an augmented extension of CIFAR-10. It contains the images from CIFAR-10 (60,000 images, 32x32 RGB pixels) and a selection of ImageNet database images (210,000 images downsampled to 32x32). It was compiled as a 'bridge' between CIFAR-10 and ImageNet, for benchmarking machine learning applications. It is split into three equal subsets - train, validation, and test - each of which contain 90,000 images. [8]

2. Literature Review

We categorize this literature review into 3 parts: papers that are applied or relevant for both challenges and then papers we applied separately for each challenge.

2.1. Papers applied for both challenges

A Survey on Image Data Augmentation for Deep Learning: According to [26], Deep Learning Models have made a tremendous progress in discriminative tasks in various domains and one of them is computer vision. This has been fulfilled by the rapid growth of deep network architectures, powerful computation, and access to big data. Deep neural networks have been effectively utilized in Computer Vision tasks, such as object recognition, image classification, and image segmentation thanks to the success of convolutional neural networks (CNNs). Currently, there are large number of studies with the purpose of improving benchmarks in computer vision tasks by designing algorithms based on deep CNNs. Improving generalization of the state-of-the-art models is one of the most difficult challenges. Models, that are poor generalization, have overfitting occurrence in the training data. Data Augmentation is one of the solutions to build appropriate deep learning models with lower validation errors as well as training errors. The augmented data will represent a more complete set of possible data points, that will minimize the gap between training set and validation set, as well as testing sets in the future.

Data augmentation will artificially enlarge the training set size by either **data warping** or **oversampling**. **Data warping** means transforming existing data (images) but preserving their labels. There are several techniques categorized as data warping augmentation, such as geometric and color transformations, random erasing, adversarial training, and neural style transfer. **Oversampling** means creating more synthetic data points and adding them to the training set. This category consists of mixing images, feature space augmentations, and generative adversarial networks (GANs).

Image Data Augmentation Techniques: In this survey, the authors listed the following techniques: geometric transformations, color space transformations, kernel filters, mixing images, random erasing, feature space augmentation, adversarial training, GAN-based augmentation, neural style transfer, and meta-learning schemes.

1. **Data Augmentations based on basic image manipulations:**
    - **Geometric transformations:** In this section, the authors described different augmentations based on geometric transformations and a number of other image processing functions. They also described these augmentations in the context of "safety" application. That means the likelihood of preserving the label post-transformation of a data augmentation method.
        - **Flipping:** Horizontal axis flipping is much more being used than flipping the vertical axis. This method has been proven effective on datasets such as CIFAR-10 and ImageNet.
        - **Color space:** Digital image data is usually represented as a tensor with dimension of height x width x color channels. A simple color augmentation method is to isolate a single-color channel, such as R, G, or B. Doing so, we could increase or decrease the brightness of an image by performing matrix operations on these RGB matrices.

  - **Cropping:** Cropping images is a practical processing step for data as images. This technique is useful when dealing with images that have mixed height and width dimensions by cropping a central patch of each image.
  - **Rotation:** Rotation augmentations are performed by rotating an image right or left on an axis between 1o and 359o. By using this method, there is a notice that as the rotation degree increases, the label of the image is no longer preserved post-transformation.
  - **Translation:** Shifting images left, right, up, or down could be a powerful method to avoid positional bias in the data.
  - **Noise injection:** Noise injection is an augmentation method that injects a matrix contains random values, which drawn from a Gaussian distribution, into image dataset. This could encourage CNNs learn more robust features.

- **Color space transformations:** Image data is represented as 3 stacked matrices, where each of size height x width. Lighting biases are the most regularly existing challenges to image recognition problems. Therefore, color space transformation is also known as photometric transformation. Amending the color distribution of images could be a great solution for lighting challenges in testing data. Image representations in datasets can also be simplified by converting the RGB matrices into a single grayscale image. There are some additional representations for digital color such as HSV (Hue, Saturation, and Value). In our approach, we mainly utilize HSV representation and grayscale image transformation.
- **Kernel Filters:** Kernel filter is a well-known technique in image processing to sharpen and blur images. The way the method works is sliding an n x n matrix across an image with either a Gaussian blur filter, which causes the image blurrier, or a high contrast horizontal or vertical edge filter, which will make the image sharper along edges. Kang *et al.* [16] made experiments with a unique kernel filter that randomly switches the pixel values in an n x n sliding window. They call this technique PatchShuffle Regularization. They achieved a 5.66% error rate on CIFAR-10 dataset compared to an error rate of 6.33% without using the technique.
- **Mixing Images:** Mixing images together by averaging their pixel values seems not a useful transformation method to a human, but it brings amazing results to classification models, conversely. Ionue [15] showed that pairing of samples could be built up into an effective augmentation approach. Ionue performed this SamplePairing Data Augmentation technique on the CIFAR-10 dataset, and it brought up an incredible result by reducing the error rate from 8.22 to 6.93%. However, this was not the final result Ionue could achieve. He/she found a better result, which was from 43.1 to 31.0% in error rate, when testing a reduced in size of the dataset, that reduced CIFAR-10 to 1,000 samples with 100 samples for each class. This achievement demonstrated that the SamplePairing technique has been suitable for limited datasets.
- **Random erasing:** Random erasing is another interesting data augmentation technique that was invented by Zhong *et al.* [36]. This method could be seen as analogous to dropout, except occurring in the input data space rather than in the network architecture. The technique works by randomly selecting an n x m patch of an image and masking it with either 0s, 255s, mean pixel values, or random values. The researcher achieved an error rate reduction from 5.17 to 4.31% on the CIFAR-10 dataset and the best patch masked values found to be random ones.

2. **Data Augmentations based on Deep Learning:** In this category, there are 5 augmentation techniques including feature space augmentation, adversarial training, GAN-based augmentation, neural style transfer, and meta-learning schemes. In our approaches, we have not used this category of technique, so we just listed them out for introductory.

Some augmentation techniques enlighten how an image classifier could be improved, while others do not. For example, it is easy to explain the advantage of horizontal flipping or random cropping. On the contrary, it is unclear why mixing images together technique such as PatchShuffle regularization or SamplePairing is so powerful. An interesting characteristic of augmentation methods is their ability to combine together. For example, the random erasing technique could be bundle up with any of other augmentation methods. An interesting question for practical data augmentation is how to determine post-augmented dataset size. There is no specific way to specify which ratio of original to final dataset size would make the model perform the best. Additionally, there is also no exact method for finding the best strategy to combine data warping and oversampling techniques. All the augmentation techniques perform

best under the assumption that training and testing datasets drawn from the same distribution.

In this project, we applied most of the data augmentations based on basic image manipulations, which will be discussed later in Section 3.

Structural Analysis and Optimization of Convolutional Neural Networks with a Small Sample Size : In [10], D'souza *et al.* conducted a throughout experiment to gain some insights on the influence of network structure on image classification performance using Convolutional Neural Networks (CNN) [23] in the small data context, with the motivation that not all image training samples are abundantly available, especially with some specific application domains.

With the speedy growth of available computation resources and Big Data era, more and more bigger, deeper models leveraging CPU, GPU and TPU show a common generalization principle that the deeper model and the more data feed in, the networks would perform better. However, those deep networks were developed under the assumption that the data samples would be sufficiently high enough, while there are huge amount of real-world applications having very limited sample size. "Small data" is consider when there are less than 5000 samples in the dataset, and some obvious applications that have access to limited data are medical images (including MRI, CT, ultrasounds, etc..).

After the experiments on VC-dimensions [32] of different model structure generated, the author defined some best network structure for small datasets such as MNIST, CIFAR-10 and gave the arguments that deeper model are not always better. Another conclusion was that the optimal model structure depends on the data's nature, not so much on the data size.

We base on some conclusion of this article to decide on our model to be used for challenge 1.

Improving multiclass classification by deep networks using DAGSVM and Triplet Loss: In [2], Agarwal *et al.* suggests a new method of improving multi-class classification accuracy, mainly by replacing the softmax function and having a triplet loss with a SVM classifier implemented using Directed Acyclic Graph (aka DAGSVM). DAG is a graph of nodes arrangement organized in a hierarchical structure, and when each node represents a SVM (One vs One binary classifier). For a problem of N classes, the DAG have $N(N-1)/2$ number of nodes, as shown in figure 1.

The author also use triplet loss to learn discriminative features before feeding into the SVM classifier, where the online triplet selection policy is to pick the negatives randomly but still make sure that the anchors in a mini-batch have the same contribution from both classes.

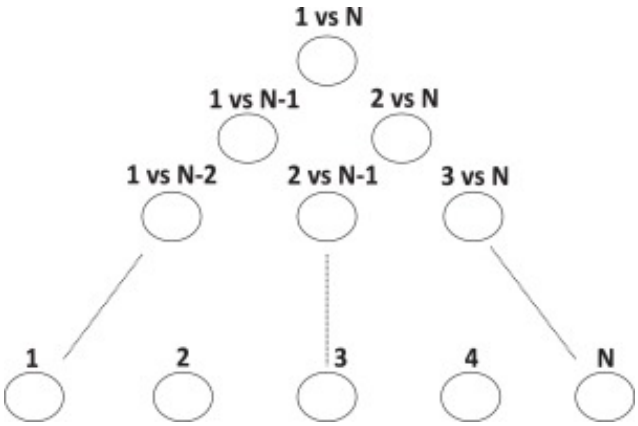


Figure 1. Directed Acycli Graph (DAG)

The experiments were performed on CIFAR-10 and STL-10 datasets, with applied augmentations like data cropping and horizontal flipping. Fine-tuning after 4k iteration, the models using DAGSVM exceeded test accuracy of the models using the same network but with Softmax output.

In the discussion part, the authors found that when improving the performance of each individual binary classifier in the DAG, the overall solution also gets positive impact. Moreover, the model can also be extended to support a large number of classes, with just the negative impact on additional training time.

In our project, one of our approaches in challenge 1 was inspired by this paper (although it can be applied for both challenges like the deep metric learning method mentioned previously). Moreover, the notable thing to mention is that the default algorithm of the SVM classifier in Scikit-learn library that we are going to implement is also based on one-vs-one policy.

2.2. Papers applied for challenge 1

Deep Learning on Small Datasets without Pre-Training using Cosine Loss: Barz *et al.* in [3], have showed that applying the cosine loss function gives considerably better performance than using cross-entropy loss after softmax activation, which is the most common choice for classification problems, on limited dataset size with a small number of samples per class. They conducted experiments on 5 small image datasets including CUB, NAB, Stanford Cars, Oxford Flowers, MIT Indoor Scenes and one text dataset which is AG News.

To address the problem of learning from limited data, there is an enormous number of studies in the field of few-shot and one-shot learning. Specifically, models will learn features from a large training dataset so that they could perform well on it, later on taking these models to perform on a dataset with limited data for each class by using nearest neighbor approach. However, Barz et al. have solved

the problem in a different way. They made a deep neural network classifier learning directly on small datasets, without pre-trained on external datasets. The cramped datasets they were training on even just have roughly between 20 and 100 data points per class. In addition to that, inspired by the work of Barz and Denzler [4], which used cosine loss to map images onto semantic class embedding derived from a hierarchy of classes in the context of image retrieval. Although the purpose of Barz and Denzler is to improve semantic consistency of image retrieval results, they also realized that classification accuracies were impressive on NAB dataset without pre-training. Barz and *et al.* in [3] have shown that the reason behind the above circumstance is the cosine loss function and this loss function could be implemented in any limited dataset size to achieve remarkable classification accuracy than using traditional cross-entropy loss, just by embedding classes with one-hot vectors.

Cosine Loss: The cosine similarity between d-dimensional vectors $a, b \in \mathbb{R}^d$ is defined as:

$$\sigma_{\cos}(a,b) = \cos(a \angle b) = \frac{\langle a,b \rangle}{\|a\|_2 \cdot \|b\|_2}, \quad (1)$$

where: $\langle \cdot,\cdot \rangle$ denotes the dot product and $\|\cdot\|_p$ denotes the L_p norm.

Let $x \in \mathfrak{X}$ be an instance in the training set (images, text, etc.) and $y \in \mathcal{C}$ be the class label of x from the set of classes $\mathcal{C} = \{1, \ldots, n\}$. In addition to that, $f_\theta : \mathfrak{X} \to \mathbb{R}^d$ denotes a transformation with model's parameters θ from the input space $\mathfrak{X}$ into a d-dimensional Euclidean feature space by a deep neural network classifier. $\psi : \mathbb{R}^d \to \mathcal{P}$ *and* $\varphi : \mathcal{C} \to \mathcal{P}$ denotes the embedding features and classes into a common prediction space $\mathcal{P}$, respectively.

Below is an example of a class representation as a one-hot vector:

$$\varphi_{onehot}(y) = \left[\underbrace{0 \ldots 0}_{y-1 \text{ times}} \quad 1 \quad \underbrace{0 \ldots 0}_{n-y \text{ times}}\right]^\top . \quad (2)$$

Barz *et al.* define the cosine loss function as:

$$\mathcal{L}_{\cos}(x,y) = 1 - \sigma_{\cos}(f_\theta(x), \varphi(y)) . \quad (3)$$

In practice, the features learned by the deep network are L^2-normalized as: $\psi(x) = \frac{x}{\|x\|_2}$. Equation (3) becomes:

$$\mathcal{L}_{\cos}(x,y) = 1 - \langle \varphi(y), \psi(f_\theta(x)) \rangle . \quad (4)$$

The L^2-normalized operation above puts a restriction on the prediction space $\mathcal{P}$ to make it in the unit hypersphere. To make the equation (4) holds, the class embeddings $\varphi(y)$ must lie on the unit hypersphere as well. One-hot vector is a special case that does not need explicitly L^2-normalized as they have unit-norm by definition.

Comparing Cosine Loss with Categorical Cross-Entropy Loss and Mean Squared Error:

- **Mean Squared Error (MSE):** MSE is the simplest loss function as it does not apply any transformations to the feature space, so it uses an Euclidean prediction space. The difference between samples and their classes in this space is measured by the squared Euclidean distance:

$$\mathcal{L}_{MSE}(x,y) = \|f_\theta(x) - \varphi(y)\|_2^2 . \quad (5)$$

  From the definition, we could know that the basic difference between cosine loss function and MSE is cosine loss function has L^2-normalized on the feature space, while MSE does not.

- **Categorical Cross-Entropy Loss:** Categorical cross-entropy loss is the most frequently used loss function in the deep learning classification problem. The dissimilarity between the prediction and true class is measured as following:

$$\mathcal{L}_{xent}(x,y) = -\langle \varphi(y), \log(\frac{\exp(f_\theta(x))}{\|\exp(f_\theta(x))\|_1}) \rangle , \quad (6)$$

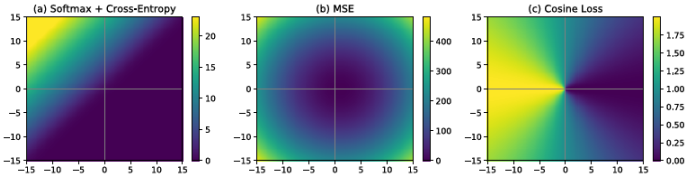


Figure 2. Heatmaps of three loss functions in a 2-D feature space with $\varphi(y) = [1 \quad 0]^\top$ (image from [3])

Barz et al. made a comparison between the 3 loss functions in a 2-dimensional feature space for the class $y = 1$ (Figure 2). From the figure, we could see that the cosine loss is bounded in $[0, 2]$, while cross-entropy and MSE have pretty high values. Additionally, cosine loss is constant even scaling the feature space because it is dependent on the direction of the feature vectors, not on the magnitude.

Finally, using cross-entropy loss is sensitive to overfitting issue, which is critical when learning with limited data. To mitigate the problem, the author mentioned *label smoothing* technique [28], which adds noise to the ground-truth distribution as regularization. In contrast to this, the cosine loss leverages the L^2 normalization to resolve the overfitting problem,

so that it does not need to introduce an additional hyper-parameter that would need to be fine-tuned for every dataset.

In [3], the researchers not only used one-hot vector class embeddings but also semantic class embeddings, that analyzes the semantic relationships between classes. From the paper, the authors state that the effect of semantic class embeddings is just complementary, so that in our challenge 1 approach we do not use this technique.

Barz *et al.* conducted a number of experiments on different small datasets such as CUB, NAB, Cars, Flowers-102, MIT Indoor, and CIFAR-100. From the author's study on effect of dataset size (defined by the number of sample per class), it is interesting to look at figure 3, where for the case of CIFAR-100 dataset classifcation with only 10 samples per class (exactly the same for our challenge 1), the cosine loss function implementation could achieve better performance compared to other loss functions.

From the perspective of the authors on solving the classification tasks on datasets with limited data and the accuracies on testing sets of similar datasets of our challenge 1, we decided to utilize the cosine loss function in challenge 1 of our project for one of our main approaches due to some similarities in the requirements of the challenge.

Deep Metric Learning: A Survey: The goal of metric learning is to measure the similarity between samples with a choice of an optimal distance for learning tasks. Inspiring by the success of Siamese and Triplet networks, Kaya *et al.* in [17] aim to provide a comprehensive study about those networks and factors that make them successful in understanding the relationships between samples including sampling selection technique, appropriate optimal distance metric, and the structure of the deep network.

Metric learning is an approach to learn from samples that introduces a new distance metric with the aim for diminishing the distance between samples in the same class and escalating the distance between samples from different classes. In other words, the technique brings similar objects closer and pushes contradictory objects far away.

The mathematics formula that illustrates the distance metric of metric learning is described as follow:

$$d_M(x_i, x_j) = \sqrt{(x_i - x_j)^\mathsf{T} \mathsf{M} (x_i - x_j)}\,,$$

where: $x_i \in \mathbb{R}^d$, $x_i \in X = [x_1, x_2, ..., x_n] \in \mathbb{R}^{d \times N}$, and N is the total number of training samples.

M is a symmetric matrix and positive semi-definite. Its values are called eigenvalues and they need to be positive or zero. Decomposing M, we have:

$$\mathsf{M} = W^\mathsf{T} W$$

$$d_\mathsf{M}(x_i, x_j) = \sqrt{(x_i - x_j)^\mathsf{T} \mathsf{M} (x_i - x_j)}$$

$$d_\mathsf{M}(x_i, x_j) = \sqrt{(x_i - x_j)^\mathsf{T} W^\mathsf{T} W (x_i - x_j)}$$

$$d_\mathsf{M}(x_i, x_j) = \|W x_i - W x_j\|_2$$

W has a linear transformation property, so that it could transform the Euclidean distance in the transformed space to a Mahalanobis distance in the original space.

Kaya *et al.* pointed out that sample selection is one of the critical factors for the success of deep metric learning in a classification or clustering problem. Triplet networks utilize triplets including an anchor, a positive, and a negative sample to be trained for classification tasks. However, in [7], the author stated that there are some poor triplets that have no effects on training the model so that they are causing waste of time and resources. Thus, it is important to choose informative triplets for deep metric learning.

There are three methods for choosing informative triplets: hard negative mining, semi-hard negative mining, and easy hard negative mining. Hard negative samples correspond to false-positive samples that are figured out from the training data. Semi-hard negatives is adopted to find negatives that are within a given margin. Lastly, easy negative mining is identifying negative samples that are beyond the anchor samples when comparing to hard negative mining. The above definitions are illustrated in figure 4. In our challenge 1, our group decided to use all of the three above triplet selection methods in other to have the achieve the performance for the triplet loss function in deep metric learning, although this can be also applied for challenge 2.

2.3. Papers applied for challenge 2

Investigation of transfer learning for image classification and impact on training sample size [37] Transfer learning is one of the practical solutions to reduce the data required for training, which tries to reuse learned knowledge for similar tasks. Using this approach, the data required for domain specific tasks can be significantly reduced. several technical details are not clearly addressed, For example, in every DL-based classification model, there are two parts inside, namely the feature extraction parts (stacking of convolutional and pooling layers) and classification parts (fully connected layers) When using transfer learning to build a classification model, it is uncertain which parts play a more important role in successfully applying to a new dataset. Additionally, there are various DL models for image classification, such as VGGNet, ResNet, and

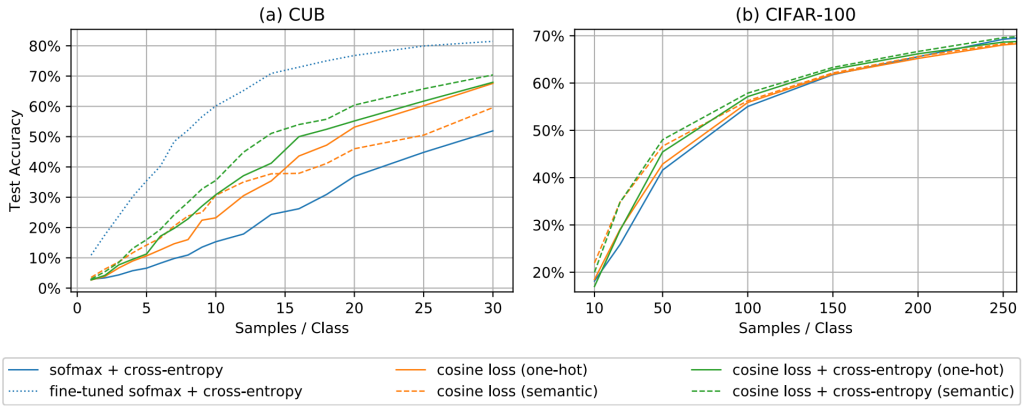
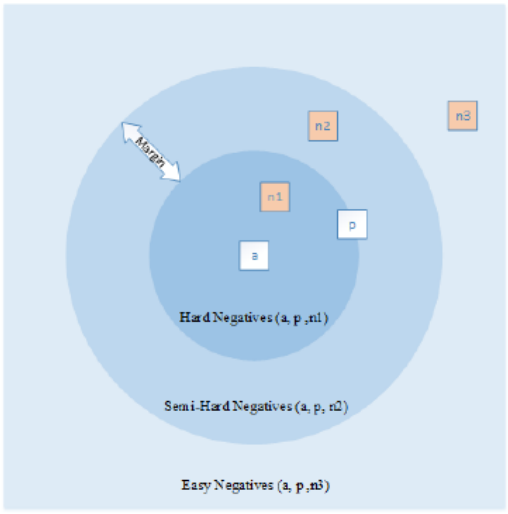


Figure 3. Classification performance (%) depending on dataset size (image from [3]). Cosine loss function's implementations gave a better classification performance for low samples per class cases (10, 50, 100, 150) compared to other loss function's implementations



Hard Negative Mining
$d(a,n) < d(a,p)$

Semi-Hard Negative Mining
$d(a,p) < d(a,n) < d(a,p) + margin$

Easy Negative Mining
$d(a,p) + margin < d(a,n)$

Figure 4. Negative Mining from [17]

others. The selection of the bare-bones model could be an important point for the implementation of transfer learning, particularly for the cases with limited data.

Deep CNN like VGG and Resnet models have about millions of parameters to tune these large amounts of model parameters, a big training dataset is necessary and require much more parameters to be adjusted than lightweight models. This study summarizes useful suggestions and guidelines for scientists and engineers with little or no machine learning backgrounds to help them develop DL-based image classification models in their respective areas.

VGGNet [35], authors proposed five configurations with different numbers of layers with trainable weights. The error rate of the model decreases as the number of layers increases, but when VGGNet reaches 19 layers, the error rate reaches saturation. After that, a degradation problem has been reported where the accuracy degrades rapidly with the increasing of the network layers so lightweight model could be more beneficial with lake of training data.

The metric-based approach is one of the most popular methods of few-shot learning that embed the input images into the feature space, where distance metrics determine the class assignment. Based on this idea, different approaches have been proposed. The Siamese network [20] uses the contrastive loss function. The cosine distance and the Euclidean distance are utilized in the matching network and prototypical network respectively.

Transfer learning Implementation follows two common approaches:

- replaces the final classification layer by a linear SVM classifier.
- replaces the final layer with a new softmax classifier

according to new problem output classes size which we used in our experiments look figure 5.

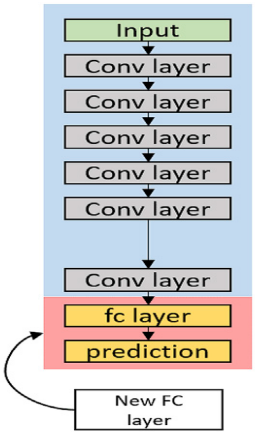


Figure 5. Transfer learning scheme [37]

The overall idea of the experiment is to use various amounts of data to train different approaches that include data augmentation DA, freezing some early layers , decide your classifier according to output size and finally calculate accuracy through the test data the following figure 6 shows results of different approaches.

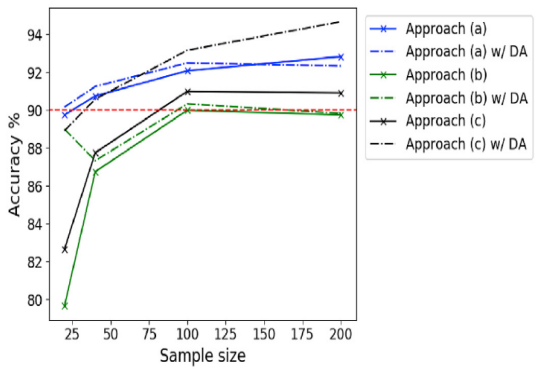


Figure 6. Transfer learning with different approaches: (a) and (b), the feature extraction parts are frozen, and only fine-tuning in the final classification portion is allowed. The main difference between approach (a) and (b) is the choice of the classifier, where a linear SVM classifier is used in approach (a), and a new softmax classifier is used for approach (b). For approach (c), the final layer is replaced first corresponding to the output size and all parameters in the network are set to be trainable. [37]

Paper Results Discussion

- freezing part of the pre-trained weights can also downgrade the model performance due to the specificity of the model.
- the large-scale model encounters training difficulties in the training when training samples are limited. The accuracy of the ResNet-152 model does not show any advantage compared with ResNet-18. Therefore, lightweight models are more suitable for fine-tuning the entire model when a sufficient amount of training data is not available.
- From figure 6 we can clearly see that data augmentation and final layer classifier are essential techniques to improve your model accuracy.
- the success of the transfer learning highly depends on the feature extraction parts of the model, and verification of the feature extractor with the target dataset is the key step before adopting transfer learning.
- DL model can achieve over 90% accuracy by using less than 100 images in each class.
- In terms of these results, *we decided to use pre-trained lightweight models like VGG16 and Resnet50 + DA and fine tuning to apply on challenge 2*.

EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks [29] Studying DL model scaling is one of the essential techniques that requires more attention to carefully balance between network depth, width, and resolution that can lead to better performance. Also, designing a new baseline network that can scale up in a systematic way would obtain a family of models that can achieve much better accuracy and efficiency than previous ConvNets.

some related work worked on model scaling problem without the concern of balanced scaling like ResNet [11] can be scaled down (e.g., ResNet-18) or up (e.g., ResNet-200) by adjusting network depth (layers), while WideResNet [33] and MobileNets [14] can be scaled by network width (input channels). It is also well-recognized that bigger input image size will help accuracy with the overhead of more FLOPS. Although prior studies have shown that network depth and width are both important for ConvNets' expressive power, it still remains an open question of how to effectively scale a ConvNet to achieve better efficiency and accuracy.
EfficientNets significantly out perform other ConvNets. In particular, EfficientNet-B7 achieves new state-of-the-art 84.3% top-1 accuracy but being 8.4x smaller and 6.1x faster than GPipe. EfficientNet-B1 is 7.6x smaller and 5.7x faster than ResNet-152. In addition being 8.4x smaller and 6.1x faster on inference than the best existing ConvNet. EfficientNets also transfer well and achieve state-of-the-art accuracy on CIFAR-100 (91.7

Rethinking the process of scaling up ConvNets, it is critical to balance all dimensions of network width/depth/resolution, and surprisingly such balance

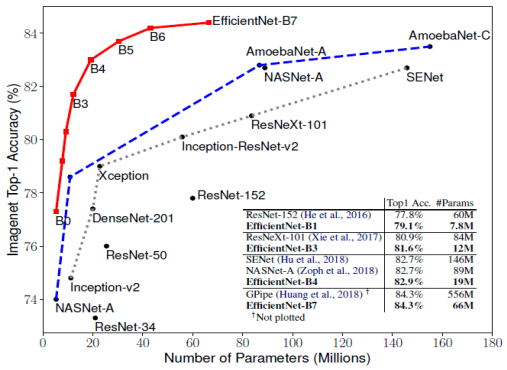


Figure 7. Model Size vs. ImageNet Accuracy. All numbers are for single-crop, single-model. EfficientNets significantly out perform other ConvNets. In particular, EfficientNet-B7 achieves new state-of-the-art 84.3% top-1 accuracy but being 8.4x smaller and 6.1x faster than GPipe. EfficientNet-B1 is 7.6x smaller and 5.7x faster than ResNet-152.[29]

can be achieved by simply scaling each of them with constant ratio. uniformly scales network width, depth, and resolution with a set of fixed scaling coefficient, the compound scaling method makes sense because if the input image is bigger, then the network needs more layers to increase the receptive field and more channels to capture more fine-grained patterns on the bigger image.

Model dimensions scaling importance

- **Depth (d)**: the deeper ConvNet can capture richer and more complex features, and generalize well on new tasks. However, deeper networks are also more difficult to train due to the vanishing gradient problem. Although several techniques, such as skip connections and batch normalization alleviate the training problem.

- **Width (w)**: Scaling network width is commonly used for small size models, wider networks tend to be able to capture more fine-grained features and are easier to train. However, extremely wide but shallow networks tend to have difficulties in capturing higher level features.

- **Resolution (r)**: With higher resolution input images, ConvNets can potentially capture more fine-grained patterns. but the accuracy gain diminishes for very high resolutions.

Scaling up any dimension of network width, depth, or resolution improves accuracy, but the accuracy gain diminishes for bigger models.In order to achieve better accuracy and efficiency, it is critical to balance all dimensions of network width, depth, and resolution during ConvNet scaling.

PAPER RESULTS & DISCUSSION

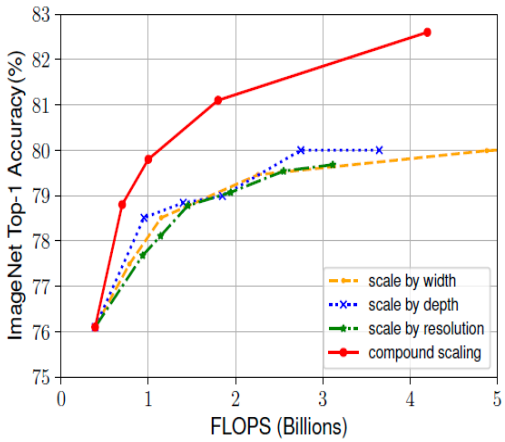


Figure 8. Scaling Up EfficientNet-B0 with Different Methods. [29]

- All scaling methods improve accuracy with the cost of more FLOPS, but compound scaling method can further improve accuracy, by up to 2.5%, than other single dimension scaling methods, suggesting

- EfficientNet models consistently reduce parameters and FLOPS by an order of magnitude (up to 8.4x parameter reduction and up to 16x FLOPS reduction) than existing ConvNets.

- EfficientNet models generally use an order of magnitude fewer parameters and FLOPS than other ConvNets with similar accuracy

- EfficientNet Performance Results on Transfer Learning Datasets achieve new state-of-the art accuracy for 5 out of 8 datasets, with 9.6x fewer parameters on average which is our most concern in this study.

- Inference latency, EfficientNet-B1 runs 5.7x faster than the widely used ResNet-152, while EfficientNet-B7 runs about 6.1x faster than GPipe.

- In terms of these results, *We decided to use EffienetNet-B0 pretrained model and compare it accuracy score with VGG16 and Resnet50 models for challenge 2.*

3. Methodology & Implementation

For this project, our team focused more on challenge 1 and did our developing works with a more practical approach rather than from theoretical standpoints (but the key decisions were still made based on commonly known knowledge in machine learning (ML) and deep learning (DL), plus some amount of paper researches). Through trial and errors, the experimental results verified what were mentioned in literature review.

3.1. Challenge 1

3.1.1 Models performance evaluation method

Based on the goals mentioned in 1.2, beside the timing constraints, we evaluate our work by determining for a model, what are the final average and standard deviation of evaluation accuracy after running at least 4-5 instances (we also call the instances as ”run” in our context) for a specific ”hyper-parameter setting” (a.k.a ”configuration”), because the small train sample size will induce randomness and variation in each different run. Moreover, in order to see how good a configuration is we also need to know how the model progresses toward the final accuracy. Thus, we based on plotting the test accuracy curves (prediction accuracy on the validation set) of all runs from a same configuration during training across epochs to determine a model's performance at that specific configuration. We applied this evaluation method for all models involving using an optimizer with gradient descent over a number of epochs, except for linear models' mentioned later that do not train with gradient descent.

Figure 9 shows the given baseline CNN model 's performance (without any modification on hyper-parameters, except the increased number of epochs to 700 to have a broader view). It can be observed that the initial baseline CNN model's test accuracy curves has very high fluctuation during ramping phase, and reaches saturation in very early epochs; two other points to be noticed are that the model's maximum accuracy points of each run are way too high compare to the final saturated accuracy level.

Thus, some criteria we defined ourselves to consider the performance of a model configuration ”stable” based on the test accuracy curves like in figure 9 are:

- Accuracy curves of different runs should not varies too much during ramping phase.

- Maximum accuracy in the training process should be only higher than the final ”stable” accuracy state by a considerably small amount.

- During final epochs when the model already reach the stable phase, the accuracy variation (or standard deviation) across runs should be as small as possible.

From a logical standpoint a model met all those 3 criteria when running on the test bed will likely to be more generalized and will have a not so worse performance (specifically on test accuracy variation) when running in the test set in CodaLab.

Our final best model in challenge 1 satisfied all of these 3 criteria.

3.1.2 Optimal model selection for CIFAR-10 dataset

As summarized previously in 2.1, from [10], the best network architecture for each dataset is unique, or data-driven,

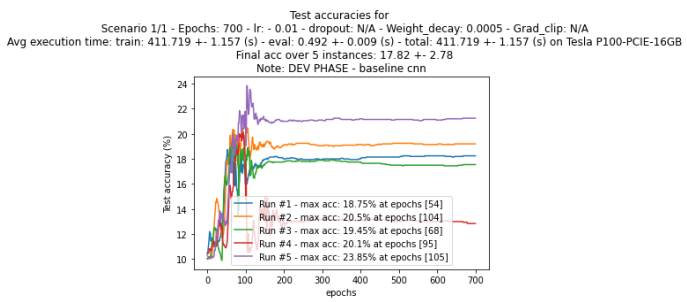


Figure 9. Performance of the given baseline CNN model without any technique applied, presented by test accuracy for each epochs across all 5 runs. Model parameters, brief summary on accuracy and execution time (with corresponding GPU device name) are also included on the plot's title and legend (although the timing is not so accurate because of the extra forward passes when running evaluation on the validation set after each epoch).

and going deeper even makes the models' performance degrade. That is the reason why for a problem of learning on limited data without pre-training like in challenge 1, we selected **ResNet-9** with only 9 residual blocks as the base model for our development.

Experiment results in 3.1.5 also verify that conclusion in the paper. We tried deeper models (such as ResNet-18, ResNet-50, DenseNet, RestNext-14) with the same configuration from our base ResNet-9 model but could not achieve similar performance, of if could they would take much longer training time and much more epochs.

Besides, as explained by Andreas *et al.* in [31], Residual Network shows the behavior of an ensemble of many shallow networks, ensemble in ML generally can help increase the model's generalization.

3.1.3 Applied techniques for all approaches

Data Augmentation: Data augmentation played a critical role in boosting the performance of our models in all approaches, especially when the data samples are limited.

Due to the requirements of challenge 1, we cannot use advance GAN method to generate more samples for training. So we just applied usual basic image manipulation transformations from PyTorch, such as: *RandomCrop, RandomGrayscale, RandomHorizontalFlip, RandomAffine, ColorJitter*.

The key in this technique base on our experiments is to find the most appropriate setting for the mentioned transformations, in which our team did by took a survey from best commonly used settings for those transforms (for example, from [24]).

Figure 10 shows how bad data augmentation bring so little improvement in the model's performance. Especially, in our final optimal configuration, the proper setting on *ColorJitter* did enhance the ResNet-9 model's average perfor-

mance at least by 5%.

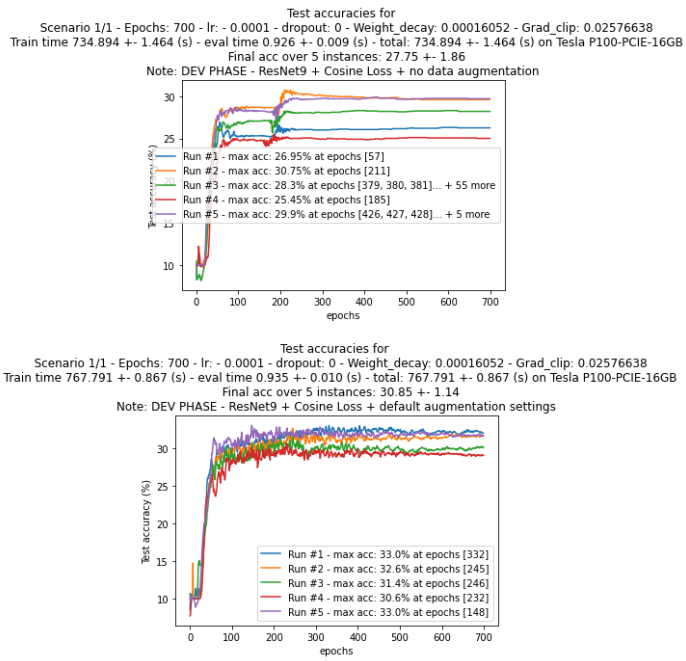


Figure 10. Effect of bad data augmentation on model performance. The image above is the plot of test accuracy curves on the ResNet9 model without any data augmentation. The image below is the plot in the case with default settings of data augmentation applied. All remaining hyper-parameter settings were kept the same with the optimal solution. For both cases, although the test accuracy trends are quite stable, but the default transform settings could not increase the ResNet9 model performance a lot, despite helping a little bit on runs variation

The best data augmentation parameters for challenge 1 is listed in Appendix section, figure 20

Adam Optimizer with weights decay and Gradient Clipping: Adam [19] is one of the most popular and powerful optimization algorithm nowadays, which can be seen as a combination of Gradient Descent with momentum and RmsProp.

On the other hand, gradient clipping [34] is one of the techniques that not only helps reduce the gradient exploding problem of deep neural network, but also accelerates training [34].

In all of our approaches using a CNN network or a part of it, we applied this combination and have weight decay and gradient clip as 2 of the hyper-parameters for model fine-tuning.

Cosine Loss function: All of our models evaluation with pure CNN approach was applied Cosine Loss function, due to the better performance in limited data situation as mentioned before in literature review [3] compared to the commonly use Cross Entropy Loss function.

Learning rate scheduler: Again, for all model using gradient descent during training, we applied a learning rate scheduler with *OneCycleLR* policy from PyTorch [27]. This learning rate scheduler acts as a regulator for learning rate, combining with the adaptive learning rate mechanism from Adam optimizer and creating the fastest convergence.

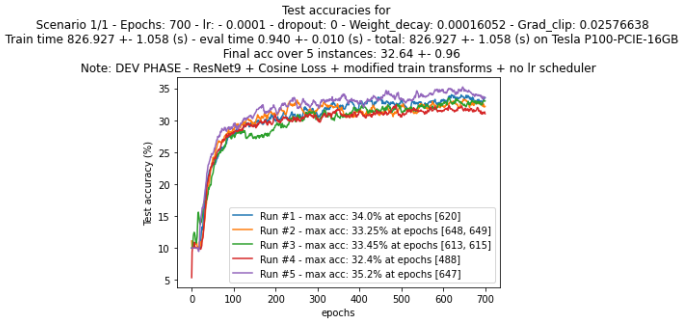


Figure 11. ResNet-9 model's performance with all best configuration except not using the learning rate scheduler. Even with the best data augmentation applied, without the help from the scheduler, the test accuracy trends still have quite high fluctuation after reaching the saturation and could not improve much phase

Grid search and Random search: Last but not least, these two are the most effective techniques in model fine tuning in Machine Learning to help find the best hyper-parameters of a model.

Initially, we performed grid search on ResNet-9 model hyper-parameters search as *learning rate, weight decay, gradient clipping, drop out and epochs* from a few dozen of setting combinations, in order to have a first few good configurations to try on other CNN models. However, grid search generally will not help getting the most optimal configuration for a model, but random search was proved to be a better method to increase the opportunity to reach better hyper-parameters settings [5].

One of the major drawbacks of using grid search is: in order to have a wide and deep enough searching range requires a huge number of configuration scenarios: Even when only we apply random search on log uniform scale for 2 parameters (weight decay and gradient clip) the minimized number of possible scenarios were already 300, not to mention the GPU time required to execute the random search and each scenario needs to execute 6-8 times (runs).

Thus, to overcome the long running time and sudden interrupts in the middle due to Google Colab's limitation on GPU usage per day, the team developed a framework to perform grid search/ random search with resuming mechanism for the next execution time. And to easily track the

performance across hundred of scenario, we also created a function and mechanism that automatically take summary of the best configurations (top 10%) in term of final test accuracy, max accuracy during training and best (minimum) final standard deviation, after each completed scenario.

From the top candidates provided by the summary function, we looked at the accuracy plots from those candidate and determine by eyes which ones really have good performance by the 3 criteria mentioned in 3.1.1.

As a result, although not having enough time to run through all 300 random search scenarios, we will be able to find the most optimal solution that help our ResNet-9 model get a good performance not only in the local test bed but also in CodaLab submission (as of this moment, 3 runs of our model in CodaLab are still at the first place of the competition dashboard).

Figure 12 not only showed how random search could help us to improve the stability of our model based on the shape of the accuracy curves, but also showed the final "convergence" of all 6 runs to a very similar test accuracy in our best model in challenge 1 (in the bottom image).

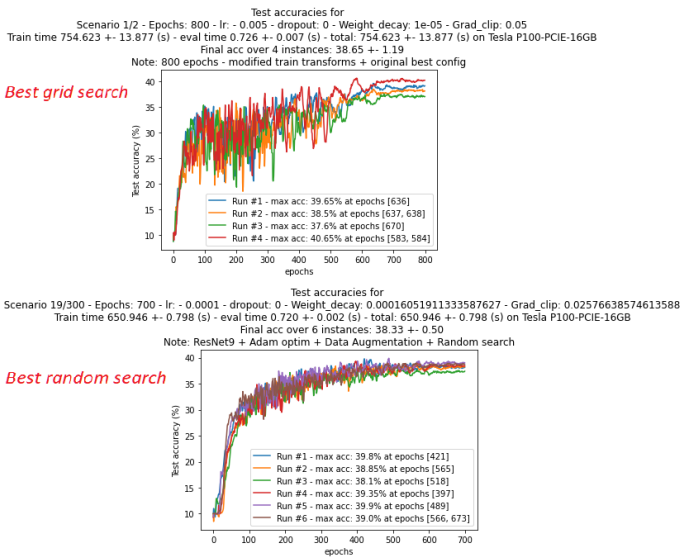


Figure 12. Improvement on ResNet-9 model performance from best grid search configuration (top) to best random search configuration (bottom). In the top image, despite reaching a similar final mean accuracy, the final accuracy standard deviation is higher than the bottom image, and more importantly, the high variation of the accuracy curves across 4 runs are much higher and hence gives hints of some uncertainty and less reliability in future runs on the test set in CodaLab.

3.1.4 Different approaches for challenge 1

The team developed 3 different approaches, there results will be summarized in section 3.1.5:

a) Traditional CNN models: This has been the main approach mentioned from the beginning of this section 3. We used the base ResNet-9 model with all of the techniques above and combined with the model performance evaluation method from the beginning. Moreover, we also tried different CNN models but could not reach the ResNet-9 performance as explained previously. Thus, we submitted the ResNet-9 model as the final versions for both test bed and CodaLab.

b) Traditional linear models: Despite the increasing deep learning field, traditional machine learning models are still proved to perform well enough in some situation so we would like to have a taste on how well they are in this limited data without pre-training challenge. 2 chosen models from the linear family are k-Nearest Neighbors (k-NN) and Support Vector Machine (SVM), because:

- k-NN: k-NN has been known for a simple yet decent linear classifier which classification task can be performed directly without any training required. The only disadvantage of k-NN is scalability since it requires high computation resource to calculate pairwise distances of the predict point to the whole computation during predicting time. However, in such a limited data constraints like in this project's challenge, the computation resource is not matter, which made k-NN worth considering. In this project we implemented our own k-NN algorithm from PyTorch and Numpy libraries.

- SVM [13]: SVM is one of the best linear classifiers so far, with the powerful kernel functions that transform the data input space into non-linear space before the maximizing margin process. SVC classifier and RandomSearchCV from Skikit-Learn together form a great combination that can tackle basic image classification tasks on MNIST or CIFAR-10 datasets.

  For both algorithms, we used mini-batch multiplier and data augmentation to increase training sample sizes. Besides this, we could not use gradient descents related technique and just fine-tuning performance by grid search or auto random search. The good thing is: both of them outperformed the given baseline CNN model.

c) Hybrid approach - Deep metric learning using Triplet loss combined with a linear classifier (k-NN/SVM): The motivation for this approach come from the course's lab and assignment. One of the characteristic of deep metric learning is its suitability for learning few samples.

Triplet loss is one of the more effective loss functions compared to pairwise (constrastive) loss.

From the idea that the distances of different class samples are pushed away from each other in embed space under the effect of the triplet loss, it will increase the chance for models like k-NN and SVM improve their prediction accuracy. Similar approaches are also seen from literature review [2, 30], but in this method the author did make the SVM model differentiable why our team method is just utilize the SVM classifier from Skikit learn.

Figure 13

Taking the good performance of the base model ResNet9, the final linear layer are removed to have the flatten layer (1028 features) to perform deep metric learning in embeded space during train time. When needed to perform training on the SVM or when need to make a prediction , the flatten layer will be connected with the SVM or k-NN classifier.

Here is a brief description of our implementation in this hybrid approach:

- For each epoch, accumulate a few mini-batches of 100 samples (each already gone through random data argumentation), the number of batches are based on the pre-defined batch multiplier.

- Generate triplets from the accumulated batch.

- Split generated triplets into smaller batch again to train with the Triplet loss and Adam optimizer (applied all of the gradient related techniques in 3.1.3).

- During each epoch, can use k-NN (developed from Pytorch) to measure the validation accuracy.

- At the final epoch, train the SVM classifier (from Scikit-learn) with the last accumulated batch using RandomSearchCV

- After training complete, can either use the k-NN or the trained SVM to classify the validation set for final evaluation. (However, it could be observed during development phase that the SVM classifier most of the time performed better than the k-NN by 3-5% when we tested the both prediction on the final epoch at the same time).

For triplet pairs generations, we follow the online mini-batch selection approach [25]. In order to balance between execution time and performance, our team mixed between random triplet selection, hard and semi-hard negative mining in the following way:

- When calling the triplet generation function for an accumulated batch input, the decision to whether perform random triplet selection or hard negative mining will be based on a random Bernoulli event (output 0 or 1) with an pre-defined probability p (i.e. for p = 0.1, for each 400 running epochs will have approximately 40 epochs applied hard negative mining).

- Due to the challenge of model collapse (loss function reach zero very early) [18], for a given hard negative mining batch we mix randomly 40% of semi-hard (i.e. for each 1000 samples in one accumulated batch, there would be around 400 samples selected with semi-hard negative criteria).

Our own hard-negative mining implementation is not optimal, which greatly impacted execution speed and could not have a good performance CodaLab submission. We also did not have enough time for fine-tuning the model, but it would be enough for the test bed version to achieve similar performance (a little bit lower) with the first approach (using purely ResNet-9 model) and still meet hard time constraints of 2 hours GPU run-time.

Thus, we still submitted test bed code for this approach but not submitted in CodaLab.

3.1.5 Result & Discussion

Table 1 summarize our performance of different models and approaches.

The provided baseline CNN model in line 3 has the worst performance in term of test accuracy mean and standard deviation. However, its execution times in Google Colab are the shortest due to the model's simplicity and no special technique applied.

Due to the different test set in CodaLab and the evaluation on CodaLab sample size is larger (5000 samples versus 2000 samples in test bed), the variation (standard deviation) in the CodaLab runs is also expected to be higher than the test bed environment.

Surprisingly, linear models could perform better than the baseline CNN model. The k-NN model performed slightly better than the baseline CNN model with quite fast execution time, while the SVM classifier had a larger margin, in the expense of longer training time due to the implemented random search algorithm.

Our best performed model "**ResNet9 + Cosine Loss**" got the best accuracy in both Test bed and CodaLab environment. And with the accuracy curve met all of our defined criteria, it's reasonable why the variation across run in CodaLab was also very minimal. Execution time was also efficient enough with around 2.5 minutes (155 secs) although that is roughly 30x slower than the given baseline CNN model.

Other deeper models that we try with the same best config could not reach the performance from the ResNet9 model, even running with much longer time and more epochs, which could be explaned looking at their performance plots in the Appendix section.

Finally, our "thinking out of the box" hybrid approach and with the model "**ResNet9 + Triplet Loss + SVM**" could reach the same performance in both mean and std accuracy of the CNN ResNet9 model, but due to the limited time in optimizing code (especially in triplet hard negative selection phase) and limited fine-tuning time, it's current execution speed is much longer (20x slower) than the

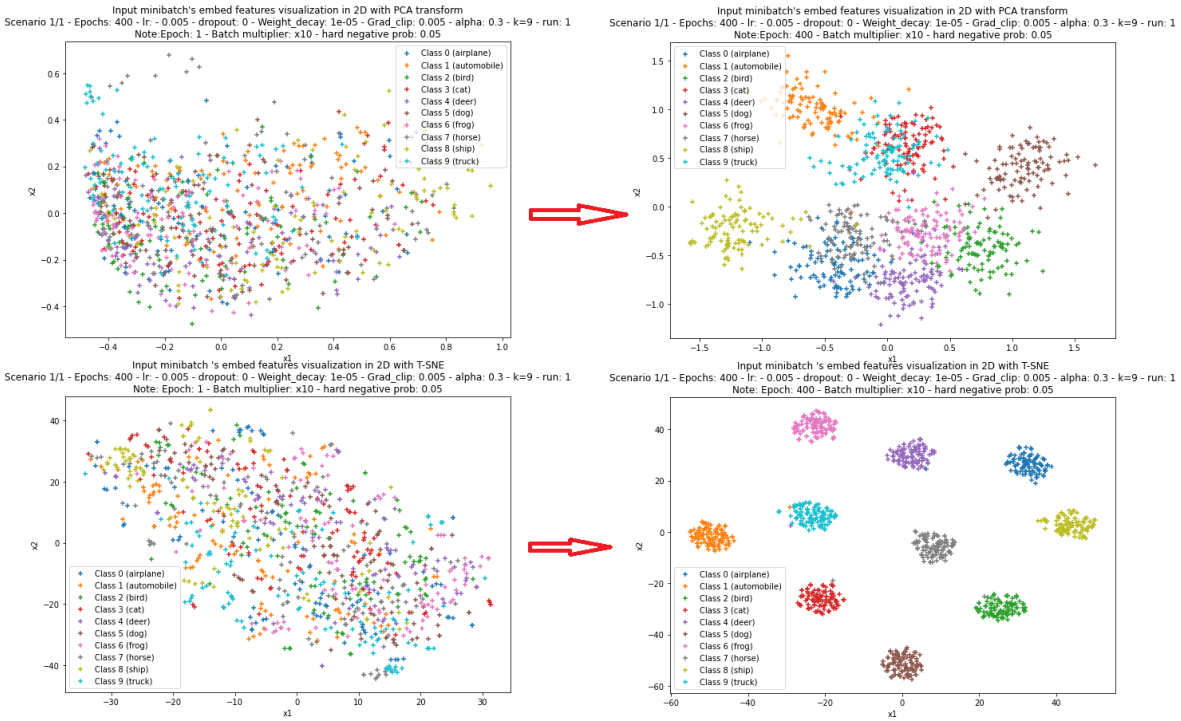


Figure 13. Visualization in 2D of input's embed space at the beging and after training completed for the hybrid approach in challenge 1. Left: distances at epoch 1 - Right: distances of after the final epoch - Top: PCA transformation - Bottom: T-SNE transformation. The triplet loss function helped push embeded points from different class away from each other, increased the classification chance for SVM, another margin-based optimization classifier.

optimized ResNet9 CNN model, and the time variation is also big across runs due to unexpected behavior of current hard negative mining algorithm. Moreover, if we replace the SVM classifier by the k-NN one (just by simply change a parameter at the beginning of the Notebook), we can reduce training time by 5-6 minutes with the trade-offs of a few percents in accuracy decrease.

3.2. Challenge 2

Transfer learning is a machine learning (ML) research problem that targets transfer knowledge gained while solving one problem and applying it to a different but related problem. This knowledge transfer would improve the performance of learning and reduce the effort to recollect the data [37]. Different deep learning models that are composed of multiple layers to learn data such as Vgg16 from [35], RestNet from [12] , Alexnet from [21] and EfficientNet from [29] can be used in advanced deep learning problems. In this challenge we are trying to use transfer learning to learn from limited data-set using different pre-trained models. we use a deep CNN with external data-set CINIC-10 [8] for training our model to learn CIFAR-10 [22] data-set classes. CINIC-10 dataset is constructed from two different sources: ImageNet and CIFAR-10. So, simply we removed all CIFAR samples from training and use only 1K samples per class for training purpose for the seek of limited the training time and resources. the best accuracy is around 86%.

The models we used are pre-trained on ImageNet dataset which contains large number of classes. the learned features from these deep models are very effective for small size datasets problems. AlexNet (Base Model), VGG16, RestNet-50 and EfficientNet-B0 are the models that we used for this challenge. For base-model, we just applied transfer of knowledge without being trained with external data. In the following sections we will discuss and analyze results.

3.2.1 Reduce Over-fitting

However, we apply feature extraction with each model and the performance is better than the baseline model (≅50%), there still exist hard overfitting due to lack of training samples. In order to reduce it, we used external dataset CINIC-10 with similar classes , implemented data augmentation and fine-tune with dropout.

Data ugmentation sample for VGG16

transforms.RandomResizedCrop(size=224,scale=(0.8,1.0)),
transforms.RandomRotation(degrees=15),
transforms.RandomHorizontalFlip(),
transforms.CenterCrop(size=224),

Table 1. Performance summary of difference approaches and model in Challenge 1. Only models with high potential were tried submitting into CodaLab. At the end, 2 models "**ResNet9 + Cosine Loss**" and "**ResNet9 + Triplet Loss + SVM**" achieved the best performance and were selected to submit as final code (although only the CNN models approach was finally submitted for CodaLab evaluation). (*): numbers measured on running during developing phase on the train set similar to the initial test bed. (**): Estimated time calculated by subtracting average execution time of the last "**ResNet9 + Triplet Loss + SVM**" model by the average training time of the SVM classifier (since both classifiers are run at the same time with the same code during final evaluation test).

Approach	Model	Test Accuracy (Test Bed)	Avg Run time (Testbed on Google Colab GPU)	Final Accuracy (CodaLab)	Final submission
Linear models	k-NN (batch multiplier x10)	$19.03 \pm 0.52\%$ (*) (over 3 instances)	≤ 20 secs	N/A	No
Linear models	SVM (batch multiplier x10)	$25.35 \pm 2.12\%$ (*) (over 3 instances)	1512 ± 4.55 secs	N/A	No
CNN models	Baseline CNN (provided)	$16.95 \pm 2.24\%$ (over 5 instances)	5.313 ± 0.036 secs	$15.64 \pm 3.13\%$ (over 6 instances)	No
CNN models	**ResNet9 + Cosine Loss**	**38.88 ± 2.86%**	154.957 ± 0.713 secs	**37.95 ± 0.35%** (over 3 instances)	**Yes** (+ CodaLab)
CNN models	ResNet18 + Cosine Loss	35.61 +- 1.02% (*) (over 4 instances)	980.714 ± 12.878 secs	N/A	No
CNN models	ResNet50 + Cosine Loss	36.08 ± 2.13 (*) (over 4 instances)	2285.981 ± 4.458 secs	$0.2999 \pm 0.0149\%$ (over 2 instances)	No
CNN models	DenseNet + Cosine Loss	29.02 ± 1.23	335.845 ± 0.605 secs	N/A	N/A
CNN models	ResNext14	$37.24 \pm 0.36\%$ (*)	927.815 ± 3.353 secs	N/A	No
Hybrid models	ResNet9 + Triplet Loss + k-NN	$36.5 \pm 3.05\%$ (over 3 instances)	3252 secs (**)	N/A	No
Hybrid models	**ResNet9 + Triplet Loss + SVM**	$36.88 \pm 2.47\%$ (over 3 instances)	3652.571 ± 620.558 secs	N/A	**Yes** (No CodaLab)

transforms.Normalize([0.485,0.456,0.406], [0.229,0.224,0.225])

Fine Tuning parameters for EfficientNet-B0:

- DROPOUT_RATE: 0.2
- LR: 0.015
- IMG_SIZE: 224
- RANDOMCROP: True
- BATCHSIZE: 64
- NUM_EPOCHS : 30
- WEIGHT_DECAY: 0.0001

3.2.2 Result Discussion

All experiments run on Google Colab Notebooks with virtual GPU. First, we download the CINIC-10 with only 1K sample/class from Github repository (link provided in notebook). Then we create training and validation directories to build the data loaders. We use accuracy and loss to decide the performance and over-fitting.

Baseline Model: For the baseline, we directly run the provided pre-trained Alexnet model with 100 training sample and 2000 validation sample without any dropout or data augmentation to be compared with the following models as shallow model. As can be seen from figure 14, the baseline performs not well about 50% validation accuracy and there exists strong over-fitting since training accuracy is much better than validation accuracy and validation loss value is greater than training loss value.

Feature extraction with models: Since we use pre-trained models, every downloaded model had modified to suites our dataset in terms of number of input channels and the output classifier number of classes. Because of limited resources, the number of training samples were bounded to CINIC-10 dataset (1K samples/class) + 10 samples /class

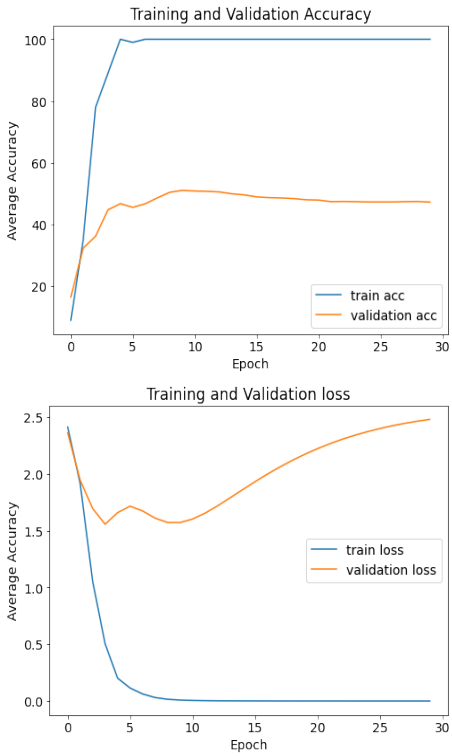


Figure 14. Baseline Model

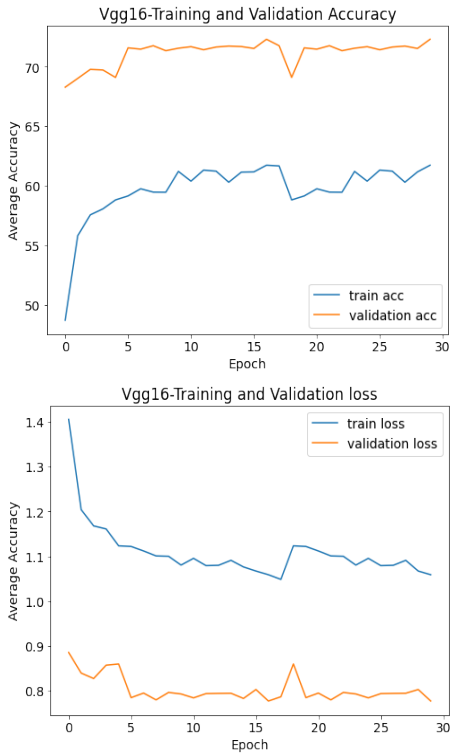


Figure 15. Vgg16 Model

from CIFAR10, the training time with GPU was under the bounded limit of 2 hours. The performance for each model is better than the baseline.

Fine-tuning with models: In order to make better performance its recommended to fine-tune these models as well. For **VGG16** we unfreeze starting model layers by setting requires_grad to False. For **EfficientNet-B0** we used pytorch implementation that cloned from github repo: https://github.com/hirotomusiker/cifar10_pytorch. We applied some modification to make it suites our dataset also some hyper-parameters have changed by help of grid search to achieve higher accuracy i.e Learning rate changed from 0.01 to 0.015, batch size from 8 to 64, WEIGHT_DECAY from 0.001 to 0.0001, IMAGE_SIZE from 128 to 224 and finally drop out from 0.3 to 0.2.

Trained Models Results: From 14, it is too fast to get the result. it takes 5̃ minutes per 30 epochs on CPU. However, the performance of this model is around 50% for validation and 100% for training. The training accuracy is much greater than validation accuracy and training loss value is much less than validation loss value. It is obvious that there exists strong overfitting since the training set size is very small (100 sample).

From 15, Vgg16 validation accuracy is greater than training accuracy, Also the training loss value is greater than validation value. Even though there is a little bit confusing but that means the model needs much training samples to generalize well which will violate the time limit allowed since VGG16 takes 65 minutes per 30 which will be doubled when increasing train data size. The validation accuracy also still near 72%.

From 16, ResNet 50 training accuracy is less than validation accuracy and training loss value is much less than validation loss value that means there still exists overfitting in the model. the model needs much training samples to generalize well which will violate the time limit allowed. ResNet-50 takes 43 minutes per 30 epochs. The validation accuracy near also still near 71%.

EfficientNets is a simple and highly effective compound scaling method, which enables us to easily scale up a baseline ConvNet to any target resource constraints in a more principled way, while maintaining model efficiency[29].

From 17, EfficientNet-B0 achieved the highest accuracy score for both training and validation in reasonable time. training accuracy is higher than validation accuracy. Also training loss value is higher than validation loss that means Model is learning well and able to generalize. the model may need much training samples and increasing number of epochs to achieve much higher accuracy. EfficientNet-B0 takes 40 minutes per 30 epochs to score validation accuracy near 86% with less amount of resource in terms of

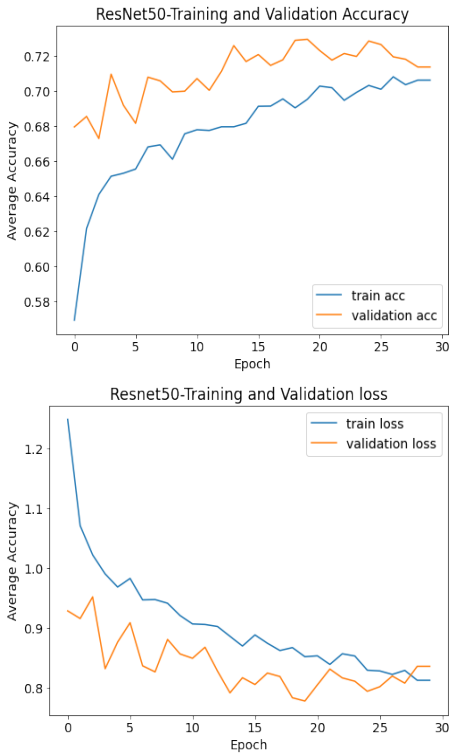


Figure 16. ResNet 50 Model

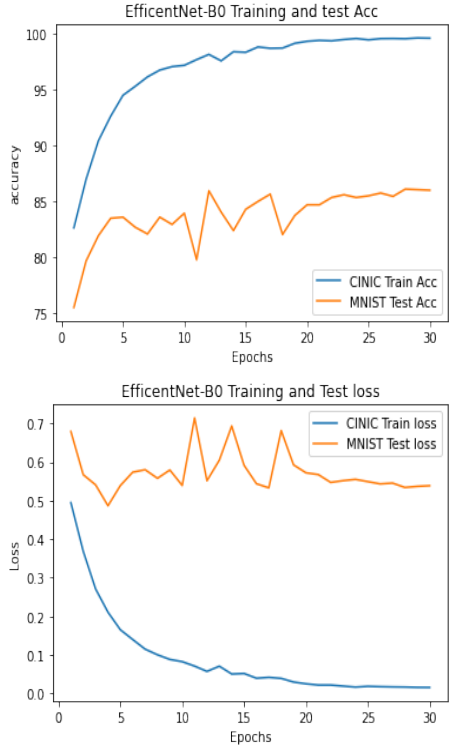


Figure 17. Google EfficentNet-B0 Model

GPU Memory. EfficientNet-B0 Model has run for 3 times for training and validation and achieve accuracy 83.77 -+ 2.30% the following figure shows how accurate the model for inference.

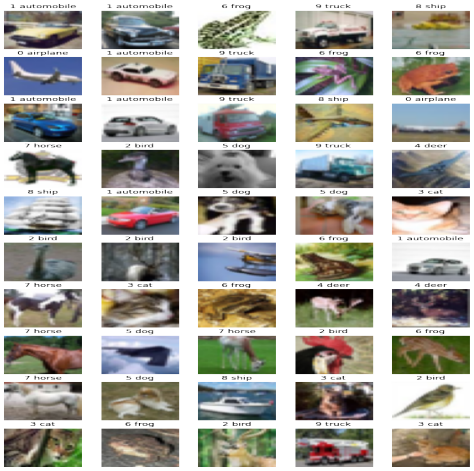


Figure 18. Efficient Net Model Inference Output

Results Summary: Table 2 summarizes the team's work on challenge 2. Our final submitted code is for model **EfficientNet B0** highlighted in **bold**

4. Conclusions

From this project, the team realized that learning from limited data is a very interesting but challenging experience.

In challenge 1, our team was able to perform well with the top result submission on CodaLab, as well as developed and implemented 2 models which give quite similar performance, although there are still things to be improve further if time allows (such as making the SVM classifier in the third approach fully differentiable as in [30]).

Learning with limited data without pre-trainning like in the challenge 1's context showed that regardless of the model and approach, the achieved performance is still below 50% test accuracy, which proved than transfer learning approaches in challenge 2 are the proper way to conquer this kind of limited data challenge (excepted when the data in new domain has too much different characteristics compared to the source domain, where pre-trained approach would not help and the challenge 1's context is un-avoidable).

For challenge 2, the objective is to show how deep models like VGG16, ResNet and EfficientNetB0 can be used on very small size data without large margin of overfitting and with great performance. All of the models are pre-trained on ImageNet. According to the experiments, most of the models had performed well, specially when apply data augmentation, dropout, and fine-tuning.

Overall, the experiments proved that deep models can

Table 2. Challenge 2 - Transfer learning models summary

Approach	Model	Train Set	Test Set	Train Accuracy	Test Accuracy (TestBed)	Run time (Testbed /Google Colab)	Final submission
Transfer learning	Baseline (AlexNet)	100 samples CIFAR-10	2K samples CIFAR-10	99%	46.25 ± 6.72% (1 CPU instance)	300 secs-Testbed	No
Transfer learning+ External Data	Vgg16	100 samples CIFAR-10 + 10K samples CINIC-10	2K samples CIFAR-10	61%	70±1.25% (1 GPU instance)	3900 secs-Colab	No
Transfer learning+ External Data	ResNet50	100 samples CIFAR-10 + 10K samples CINIC-10	2K samples CIFAR-10	70%	67±2.9% (1 GPU instance)	2580 secs-Colab	No
Transfer learning+ External Data	**EfficientNet B0**	100 samples CIFAR-10 + 10K samples CINIC-10	2K samples CIFAR-10	99%	83.77 ± 2.30% (1 GPU instance)	2400 secs-Colab	**Yes**

be used to generalize well very small datasets with proper modifications and limited resources.

References

[1] CodaLab competition for Challenge 1 - COMP 691 Deep Learning course project. https://competitions.codalab.org/competitions/30481, Winter 2021. 1

[2] Nakul Agarwal, Vineeth N Balasubramanian, and C.V. Jawahar. Improving multiclass classification by deep networks using dagsvm and triplet loss. *Pattern Recognition Letters*, 112:184–190, 2018. 4, 13

[3] Björn Barz and Joachim Denzler. Deep learning on small datasets without pre-training using cosine loss. *CoRR*, abs/1901.09054, 2019. 4, 5, 6, 7, 11

[4] Björn Barz and Joachim Denzler. Hierarchy-based image embeddings for semantic image retrieval. In *2019 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 638–647, 2019. 5

[5] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(10):281–305, 2012. 11

[6] CIFAR-10. Cifar-10 — Wikipedia. "https://en.wikipedia.org/wiki/CIFAR-10", 2021. 2

[7] Yin Cui, Feng Zhou, Yuanqing Lin, and Serge Belongie. Fine-grained categorization and dataset bootstrapping using deep metric learning with humans in the loop, 2016. 6

[8] Luke Nicholas Darlow, Elliot J. Crowley, Antreas Antoniou, and Amos J. Storkey. CINIC-10 is not imagenet or CIFAR-10. *CoRR*, abs/1810.03505, 2018. 2, 14

[9] Jia Deng, Alexander C Berg, Sanjeev Satheesh, Hao Su, Aditya Khosla, and Li Fei-Fei. Imagenet large scale visual recognition challenge (ilsvrc) 2012, 2012. 1

[10] Rhett N. D'souza, Po-Yao Huang, and Fang-Cheng Yeh. Structural Analysis and Optimization of Convolutional Neural Networks with a Small Sample Size. *Scientific Reports*, 10(1):834, Jan. 2020. 4, 10

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. 8

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016. 14

[13] M.A. Hearst, S.T. Dumais, E. Osuna, J. Platt, and B. Scholkopf. Support vector machines. *IEEE Intelligent Systems and their Applications*, 13(4):18–28, 1998. 12

[14] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017. 8

[15] Hiroshi Inoue. Data augmentation by pairing samples for images classification, 2018. 3

[16] Guoliang Kang, Xuanyi Dong, Liang Zheng, and Yi Yang. Patchshuffle regularization, 2017. 3

[17] Mahmut Kaya and H. Bilge. Deep metric learning: A survey. *Symmetry*, 11:1066, 08 2019. 6, 7

[18] Gábor Kertész. Metric embedding learning on multi-directional projections. *Algorithms*, 13(6), 2020. 13

[19] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017. 11

[20] Gregory Koch, Richard Zemel, and Ruslan Salakhutdinov. Siamese neural networks for one-shot image recognition. In *ICML deep learning workshop*, volume 2. Lille, 2015. 7

[21] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014. 14

[22] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research). 2, 14

[23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, page 1097–1105, Red Hook, NY, USA, 2012. Curran Associates Inc. 2, 4

[24] Programcreek. Python torchvision.transforms.colorjitter() examples. "https://www.programcreek.com/python/example/117697/torchvision.transforms.ColorJitter". 10

[25] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. *CoRR*, abs/1503.03832, 2015. 13

[26] Connor Shorten and Taghi M. Khoshgoftaar. A survey on Image Data Augmentation for Deep Learning. *Journal of Big Data*, 6(1):60, Dec. 2019. 2

[27] Leslie N. Smith and Nicholay Topin. Super-convergence: Very fast training of residual networks using large learning rates. *CoRR*, abs/1708.07120, 2017. 11

[28] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2818–2826, 2016. 5

[29] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114. PMLR, 2019. 8, 9, 14, 16

[30] Yichuan Tang. Deep learning using support vector machines. *CoRR*, abs/1306.0239, 2013. 13, 17

[31] Andreas Veit, Michael J. Wilber, and Serge J. Belongie. Residual networks are exponential ensembles of relatively shallow networks. *CoRR*, abs/1605.06431, 2016. 10

[32] Wikipedia. Vapnik–chervonenkis dimension. "https://en.wikipedia.org/wiki/Vapnik-Chervonenkis_dimension", 2021. 4

[33] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016. 8

[34] Jingzhao Zhang, Tianxing He, Suvrit Sra, and Ali Jadbabaie. Why gradient clipping accelerates training: A theoretical justification for adaptivity, 2020. 11

[35] Xiangyu Zhang, Jianhua Zou, Kaiming He, and Jian Sun. Accelerating very deep convolutional networks for classification and detection. *IEEE transactions on pattern analysis and machine intelligence*, 38(10):1943–1955, 2015. 7, 14

[36] Zhun Zhong, Liang Zheng, Guoliang Kang, Shaozi Li, and Yi Yang. Random erasing data augmentation, 2017. 3

[37] Wenbo Zhu, Birgit Braun, Leo H. Chiang, and Jose A. Romagnoli. Investigation of transfer learning for image classification and impact on training sample size. *Chemometrics and Intelligent Laboratory Systems*, 211:104269, 2021. 6, 8, 14

Appendix

.1. Member contributions

- Hussein: Initial ResNet9 model selection and first dev Notebook on Challenge 1; Developed and finished models in challenge 2; Meeting minutes taker.
- Tuan: Implemented grid search and random search resuming mechanism and report features for challenge 1; Tried linear classifiers (2nd approach) in challenge 1; Implemented Triplet loss with k-NN/SVM classifier for approach 3; Roof reading on Challenge 2's submitted code; Meeting facilitators.
- Dat: Fine-tuning ResNet9 models on challenge 1 using the provide grid and random search from Tuan; Explore deeper models' performance on challenge 1; Final code evaluation for challenge 2.

.2. Additional images

Results

#	User	Entries	Date of Last Entry	Team Name	Accuracy ▲
1	littleghost1712	56	05/02/21	DL_DT_Explore	0.384200 (1)
2	yangbo	13	05/03/21	DL_BO_DJ	0.361000 (2)
3	wegotnull	43	04/26/21		0.351800 (3)

54	0.3782	baseline_Resnet9_rev1.7.1.zip	05/02/2021 18:56:51	Finished	
55	0.3842	baseline_Resnet9_rev1.7.1.zip	05/02/2021 18:57:13	Finished	✔
56	0.376	baseline_Resnet9_rev1.7.1.zip	05/02/2021 18:57:32	Finished	

Figure 19. CodaLab competition dashboard for challenge 1. Our team is **DL_DT_Explore** and below table are the results of 3 run for our final submission

```
transforms.RandomCrop(32, padding=4, padding_mode='reflect'),
transforms.RandomGrayscale(),
transforms.RandomHorizontalFlip(),
torchvision.transforms.RandomAffine(degrees=30),
transforms.ColorJitter(brightness=0.4, contrast=0.4, saturation=0.4, hue=0.2),
```

Figure 20. Best data augmentation settings for challenge 1

```
Current performance summary over 85 completed scenario(s):
 - Top 9 final average accuracy:
   + Scenario: 69 - Final average accuracy: 39.35 +- 1.48%
   + Scenario: 59 - Final average accuracy: 38.97 +- 1.24%
   + Scenario: 55 - Final average accuracy: 39.06 +- 0.91%
   + Scenario: 51 - Final average accuracy: 39.31 +- 0.91%
   + Scenario: 54 - Final average accuracy: 39.06 +- 1.74%
   + Scenario: 56 - Final average accuracy: 38.98 +- 1.43%
   + Scenario: 60 - Final average accuracy: 39.23 +- 1.38%
   + Scenario: 72 - Final average accuracy: 39.06 +- 1.26%
   + Scenario: 74 - Final average accuracy: 39.14 +- 1.58%
 - Top 9 max accuracy:
   + Scenario: 57 - run 4 - epoch 670 - Max accuracy: 41.60
   + Scenario: 55 - run 6 - epoch 516 - Max accuracy: 41.70
   + Scenario: 23 - run 6 - epoch 479 - Max accuracy: 41.85
   + Scenario: 23 - run 6 - epoch 478 - Max accuracy: 41.60
   + Scenario: 23 - run 6 - epoch 477 - Max accuracy: 41.85
   + Scenario: 23 - run 6 - epoch 476 - Max accuracy: 41.75
   + Scenario: 23 - run 6 - epoch 459 - Max accuracy: 41.65
   + Scenario: 23 - run 6 - epoch 457 - Max accuracy: 41.75
   + Scenario: 70 - run 1 - epoch 487 - Max accuracy: 41.65
 - Top 9 least final accuracy variation:
   + Scenario: 2 - Final average accuracy: 38.41 +- 0.73%
   + Scenario: 17 - Final average accuracy: 38.74 +- 0.67%
   + Scenario: 19 - Final average accuracy: 38.33 +- 0.50%
   + Scenario: 25 - Final average accuracy: 38.60 +- 0.77%
   + Scenario: 34 - Final average accuracy: 35.09 +- 0.69%
   + Scenario: 62 - Final average accuracy: 38.93 +- 0.83%
   + Scenario: 68 - Final average accuracy: 38.49 +- 0.75%
   + Scenario: 76 - Final average accuracy: 38.69 +- 0.73%
   + Scenario: 78 - Final average accuracy: 38.63 +- 0.73%
```

Figure 21. Performance tracking and summary result output (used for grid search or random search). Our best performance settings was from scenario 9 from the summary

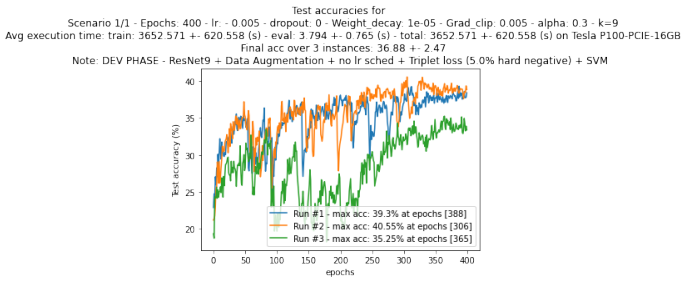


Figure 22. Performance of the hybrid approach: **ResNet9 + Triplet Loss + SVM** . We did not have enough time for fine-tuning this model

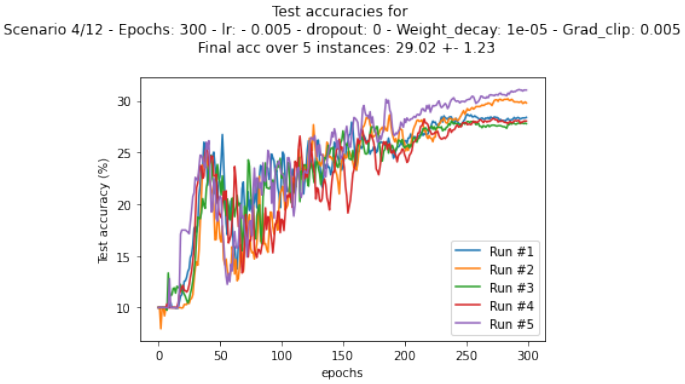


Figure 23. Best performance of DenseNet model

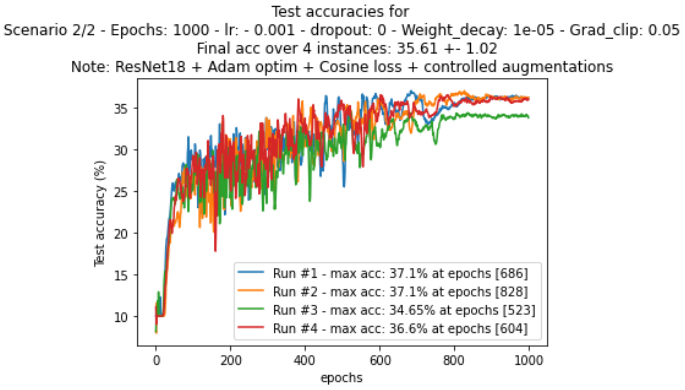


Figure 24. Best performance of ResNet-18 model

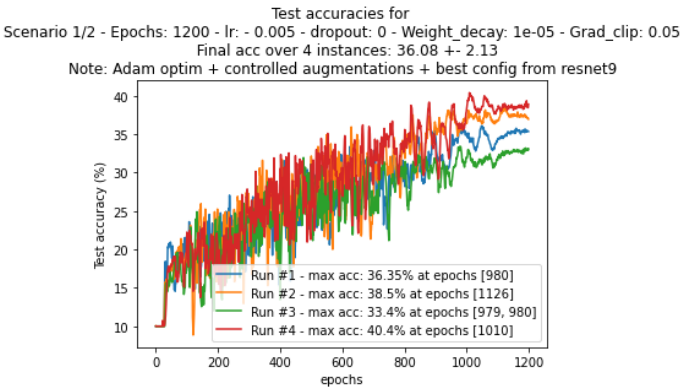


Figure 25. Best performance of ResNet-50 model

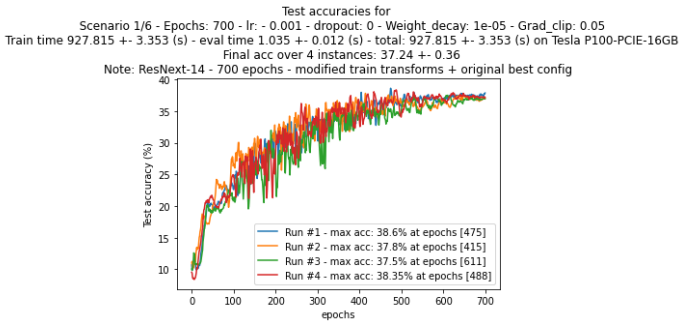


Figure 26. Best performance of ResNext-14 model

COMP 691 - Deep Learning project - Challenge 1

Team: DL_DT_Explore

Members:

- Trong-Tuan Tran
- Hussein Abdallah
- Manh-Quoc-Dat Le

This Jupyter Notebook implements the approach: ResNet9 model with Cossine Loss function.

Here the goal is to train on 100 samples. In this preliminary testbed the evaluation will be done on a 2000 sample validation set. Note in the end the final evaluation will be done on the full CIFAR-10 test set as well as potentially a separate dataset. The validation samples here should not be used for training in any way, the final evaluation will provide only random samples of 100 from a datasource that is not the CIFAR-10 training data.

Initial configurations & hyperparameters used for grid search (params defined in lists [ ] will be used for grid search):

In [23]:

```
import time
from numpy.random import RandomState
import torchvision
import numpy as np
import torch
import torch.optim as optim
from torch.utils.data import Subset
from torchvision import datasets, transforms

# Epochs: 300 - lr: - 0.001 - dropout: 0 - Weight_decay: 1e-05 - Grad_clip: 0.005
# Scenario 17/300 - Epochs: 700 - lr: - 0.0001 - dropout: 0 - Weight_decay: 0.00016051911333587627 - Grad_clip: 0.015119336467640998
# Scenario 19/300 - Epochs: 700 - lr: - 0.0001 - dropout: 0 - Weight_decay: 0.00016051911333587627 - Grad_clip: 0.02576638574613588

epochs_list = [700]
grad_clips = [0.02576638574613588] # Gradient clipping
weight_decays = [0.00016051911333587627] # Weight decay for Adam Optimizer
lrs = [0.0001] # Learning rates
drop_outs = [0] # Value for drop out layers
batch_size = 128
runs = 5 # Number of instances to run the train and test to evaluate mean and std dev of test accuracy
epoch_display_range = 100
search_plot = True # Define whether to run evaluation after each epochs to plot accuracy trend or not
log_enabled = False # Define if log down progress to support resume from previous completed run or not (for grid search)
save_image = True # If search_plot is True, enable this will save the plotted image to the img_path define below
google_drive_mount = True # If running in Google Colab and enable this, progress and image files will save to Google Drive instead
final_eval = True # Whether to run using train set or test set for final evalutation & submission
```

```
eval_str = 'DEV PHASE - ' if not final_eval else 'FINAL EVAL - '
comment = eval_str + 'ResNet9 + Cosine Loss + modified train transforms + random search best config' # comment string to put in acc
if final_eval:
    search_plot = False
    save_image = False
    google_drive_mount = False
    log_enabled = False

use_cuda = torch.cuda.is_available()
device = torch.device("cuda" if use_cuda else "cpu")
#device = torch.device('cpu')
```

The section below detects which environment is running (Colab, Kaggle or local computer). The output folders will be determine accordingly. If running in final_eval mode, no output file will be generated.

In [24]:

```
import os
output_path = 'output_txt/'
img_path = 'img/'
Colab = False
Kaggle = 'kaggle' in os.getcwd()
root = '.' # Root to download dataset
if 'google.colab' in str(get_ipython()):
    print('Running on CoLab')
    Colab = True
    from google.colab import drive
    if not os.path.exists('/content/drive/MyDrive/') and google_drive_mount:
        drive.mount('/content/drive', force_remount=False)

    else:
        if google_drive_mount:
            print('Drive already mounted at at /content/drive')

    Google_path = '/content/drive/MyDrive/Colab Notebooks/COMP691_project/' if google_drive_mount else '/'
    if not os.path.exists(Google_path):
        os.mkdir(Google_path)
    img_path = Google_path + img_path
    if not os.path.exists(img_path):
        os.mkdir(img_path)
    output_path = Google_path + output_path
else:
    if Kaggle:
        root = '../input/cifar10'
        output_path = ''
        img_path = ''
        print('Running in Kaggle')
    else:
```

```
        print('Not running on CoLab or Kaggle')
output_file_name = 'report_ADAM_cosine_improve_FINAL.txt'
output_file_path = output_path + output_file_name
progress_file = output_path + 'grid_search_progress_FINAL.txt'
img_file_name_prefix = output_file_name.replace('.txt', '')
img_file_path = img_path + img_file_name_prefix + '/'

if not final_eval:
    if not os.path.exists(img_path):
        os.mkdir(img_path)

    if not os.path.exists(img_file_path):
        os.mkdir(img_file_path)

    if not os.path.exists(output_path):
        os.mkdir(output_path)
```

```
Running on CoLab
```

Setup training/testing and other helper functions

In [25]:

```
import gc
from matplotlib import pyplot as plt

def train(model, device, train_loader, optimizer, epoch, grad_clip=None, sched=None, display=True):
    model.train()
    loss_function = nn.CosineEmbeddingLoss()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data).to(device)

        GT=torch.zeros((len(target),10))
        for idx in range(len(target)):
            GT[idx][target[idx]]=1

        GT=GT.to(device)

        loss = loss_function(output, GT, torch.Tensor(output.size(0)).to(device).fill_(1.0))
        #loss = F.cross_entropy(output, target)
        loss.backward()
        if grad_clip:
            nn.utils.clip_grad_value_(model.parameters(), grad_clip)
        optimizer.step()
        if sched:
            sched.step()
        if display and (batch_idx == 0 or batch_idx + 1 == len(train_loader)):
          print('   Train Epoch: {} [step {}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
```

```
                epoch + 1, batch_idx + 1, len(train_loader),
                100. * batch_idx / len(train_loader), loss.detach().item()))
        if device == torch.device('cuda'):
            del loss, output
            gc.collect()
            torch.cuda.empty_cache()

def test(model, device, test_loader, display=True):
    model.eval()
    test_loss = 0
    correct = 0
    loss_function = nn.CosineEmbeddingLoss()
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            GT=torch.zeros((len(target),10))
            for idx in range(len(target)):
                GT[idx][target[idx]]=1

            GT=GT.to(device)

            test_loss += loss_function(output, GT, torch.Tensor(output.size(0)).to(device).fill_(1.0)).item() # sum up batch loss
            #test_loss += F.cross_entropy(output, target, size_average=False).item()
            pred = output.max(1, keepdim=True)[1] # get the index of the max log-probability
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)
    if display:
        print('   Test set: Average loss: {:.4f}, Accuracy: {}/{} ({:.2f}%)'.format(
            test_loss, correct, len(test_loader.dataset),
            100. * correct / len(test_loader.dataset)))
    return 100. * correct / len(test_loader.dataset)

def plot_accs(accs_accross_runs, display_str, comment='N/A', save_img=True):
    plt.figure();
    #epochs = list(range(1, len(accs_accross_runs[0]) + 1))
    max_acc_display = ''
    for i, run_accs in enumerate(accs_accross_runs):
        max_acc = max(run_accs)
        max_epochs = [index + 1 for index, acc in enumerate(run_accs) if acc == max_acc]
        if len(max_epochs) > 3:
            not_display_count = len(max_epochs) - 3
            max_epochs = str(max_epochs[:3]) + f'... + {not_display_count} more'

        max_acc_display = f' - max acc: {max_acc}% at epochs {max_epochs}'
        plt.plot(run_accs, label=f'Run #{i + 1}' + max_acc_display)
    plt.xlabel('epochs')
```

```
    plt.ylabel('Test accuracy (%)')
    plt.legend();
    plt.title(f'Test accuracies for \n{display_str}Note: {comment}');

    if save_img:
        scenario = display_str[display_str.index(' ') + 1: display_str.index('/')]
        img_name = img_file_path + f'{scenario}' + generate_image_suffix()
        plt.savefig(img_name, bbox_inches='tight')

def generate_image_suffix():
    return f'_{time.time()%10000000:.0f}' + '.png'
```

Definition of ResNet9 model:

In [26]:

```
import torch.nn as nn
import torch.nn.functional as F
num_classes = 10
in_channels = 3

def conv_block(in_channels, out_channels, drop_out=0, pool=False):
    layers = [nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1),
              nn.BatchNorm2d(out_channels),
              nn.ReLU(inplace=True), nn.Dropout(drop_out)
              ]
    if pool: layers.append(nn.MaxPool2d(2))
    return nn.Sequential(*layers)

class NET(nn.Module):
    def __init__(self, in_channels, num_classes, drop_out):
        super().__init__()

        self.conv1 = conv_block(in_channels, 64, drop_out)
        self.conv2 = conv_block(64, 128, drop_out, pool=True)
        self.res1 = nn.Sequential(conv_block(128, 128, drop_out), conv_block(128, 128, drop_out))
        self.dropout = nn.Dropout(drop_out)
        self.conv3 = conv_block(128, 256, drop_out, pool=True)
        self.conv4 = conv_block(256, 512, drop_out, pool=True)
        self.res2 = nn.Sequential(conv_block(512, 512, drop_out), conv_block(512, 512, drop_out))
        self.conv5 = conv_block(512, 1028, drop_out, pool=True)
        self.res3 = nn.Sequential(conv_block(1028, 1028, drop_out), conv_block(1028, 1028, drop_out))

        self.classifier = nn.Sequential(nn.MaxPool2d(2),
                                        nn.Flatten(),
                                        nn.Linear(1028, num_classes))

    def forward(self, xb):
        out = self.conv1(xb)
```

```
        out = self.conv2(out)
        out = self.res1(out) + out
        out = self.conv3(out)
        out = self.dropout(out)
        out = self.conv4(out)
        out = self.dropout(out)
        out = self.res2(out) + out
        out = self.conv5(out)
        out = self.res3(out) + out
        out = self.classifier(out)
        return out
```

The below tries a numbers of random problem instances defined in `runs` variable at the beginning

In [27]:

```
%%time

device_name = torch.cuda.get_device_name(0) if device == torch.device('cuda') else 'cpu'

scenario_count = len(epochs_list) * len(weight_decays) * len(lrs) * len(drop_outs) * len(grad_clips)

# Statistic for CIFAR-10 datasets
normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                 std=[0.229, 0.224, 0.225])
# Data augmentation during training:
transform_train = transforms.Compose([
                                    transforms.RandomCrop(32, padding=4, padding_mode='reflect'),
                                    transforms.RandomGrayscale(),
                                    transforms.RandomHorizontalFlip(),
                                    torchvision.transforms.RandomAffine(degrees=30),
                                    transforms.ColorJitter(brightness=0.4, contrast=0.4, saturation=0.4, hue=0.2),
                                    transforms.ToTensor(),
                                    normalize]) #careful to keep this one same
transform_val = transforms.Compose([transforms.ToTensor(), normalize])


print('Running on {}'.format(device_name))

##### Cifar Data
run_on_train_set = not final_eval
dataset = 'train' if run_on_train_set else 'test'
print(f'Using CIFAR-10 {dataset} set')
print(comment)
cifar_data = datasets.CIFAR10(root='.', train=run_on_train_set, transform=transform_train, download=True)

#We need two copies of this due to weird dataset api
cifar_data_val = datasets.CIFAR10(root='.', train=run_on_train_set, transform=transform_val, download=True)
```

```
training_done = False
count = 1
scenario = 1
next_run = 1
previous_runs_accs = []
previous_train_times = []
previous_eval_times = []
previous_exec_times = []
ran_in_middle = False

if log_enabled:
    if os.path.exists(progress_file):
        with open(progress_file, 'r') as file_read:
            progress_content = file_read.readlines()
        # print(progress_content)
        if progress_content[0].replace('\n', '') == output_file_name:
            previous_scenario = progress_content[1].replace('\n', '')
            previous_runs_accs = eval(progress_content[2].replace('\n', ''))
            previous_runs = len(previous_runs_accs)
            print(f'Previous progress on {output_file_name} stopped at scenario {previous_scenario}/{scenario_count}' +\
                 f', run {previous_runs}/{runs}')
            if previous_runs == runs: # Already complete the previous scenario
                scenario = int(previous_scenario) + 1

            else: # Previous scenario just completed partially, resume in the next run
                ran_in_middle = True
                scenario = int(previous_scenario)
                next_run = previous_runs + 1
                previous_execution_times = eval(progress_content[3].replace('\n', ''))

                for i, previous_execution_time in enumerate(previous_execution_times):
                    previous_train_times.append(previous_execution_time[0])
                    previous_eval_times.append(previous_execution_time[1])
                    previous_exec_times.append(previous_execution_time[2])
            if scenario > scenario_count:
                training_done = True
                print('Training was already done!')
            else:
                print(f'Will resume training at scenario: {scenario}, run# {next_run}')

if not training_done:
    for epochs in epochs_list:
        for lr in lrs:
            for drop_out in drop_outs:
                for weight_decay in weight_decays:
                    for grad_clip in grad_clips:
```

```
                    if not ran_in_middle:

                        accs = []
                        train_times = []
                        evaluation_times = []
                        total_times = []
                        run_execution_times = []
                    else:
                        if count < scenario:
                            count += 1
                            continue #skip until reaching the scenario to run
                        accs = previous_runs_accs
                        train_times = previous_train_times
                        evaluation_times = previous_eval_times
                        total_times = previous_exec_times
                        run_execution_times = previous_execution_times
                    #scenario += 1

                    print('\nScenario %d/%d - Epochs: %d - lr: - %s - dropout: %s - Weight_decay: %s - Grad clip: %s'%(
                        scenario, scenario_count, epochs, lr, drop_out, weight_decay, grad_clip
                    ))
                    accs_accross_runs_plot = []
                    for seed in range(next_run, runs + 1):
                        start_time = time.time()
                        # Extract a subset of 100 (class balanced) samples per class for training and 2000 samples for validatio
                        permute_range = 5000 if run_on_train_set else 1000
                        prng = RandomState(seed)
                        random_permute = prng.permutation(np.arange(0, permute_range))
                        indx_train = np.concatenate([np.where(np.array(cifar_data.targets) == classe)[0][random_permute[0:10]] f
                        indx_val = np.concatenate([np.where(np.array(cifar_data_val.targets) == classe)[0][random_permute[10:210


                        train_data = Subset(cifar_data, indx_train)
                        val_data = Subset(cifar_data_val, indx_val)

                        print('  Run# [%d/%d] - Num Samples For Training %d - Num Samples For Val %d'%(seed, runs, train_data.in

                        train_loader = torch.utils.data.DataLoader(train_data,
                                                                   batch_size=batch_size,
                                                                   shuffle=True)

                        val_loader = torch.utils.data.DataLoader(val_data,
                                                                 batch_size=batch_size,
                                                                 shuffle=False)

                        model = NET(in_channels, num_classes, drop_out)
                        model.to(device)
```

```
            optimizer = torch.optim.Adam(model.parameters(),
                                         lr=lr,
                                         #momentum=0.9,
                                         weight_decay=weight_decay)
            sched = torch.optim.lr_scheduler.OneCycleLR(optimizer, lr, epochs=epochs,
                                                        steps_per_epoch=len(train_loader))
            test_accs = []
            eval_time = 0
            for epoch in range(epochs):
                print_condition = epoch%epoch_display_range==0 or epoch==epochs-1
                train(model, device, train_loader, optimizer, epoch, grad_clip=grad_clip,
                    sched=sched, display=print_condition)
                if search_plot:
                    eval_start = time.time()
                    test_acc = test(model, device, val_loader, display=print_condition)
                    eval_time = time.time() - eval_start
                    test_accs.append(test_acc)


            train_time = time.time() - start_time
            train_times.append(train_time)
            final_eval_start = time.time()
            final_acc = test_accs[-1] if search_plot else test(model, device, val_loader)
            accs.append(final_acc)
            final_eval_time = eval_time if search_plot else time.time() - final_eval_start
            evaluation_times.append(final_eval_time)
            if search_plot:
                accs_accross_runs_plot.append(test_accs)
            total_time = time.time() - start_time
            total_times.append(total_time)
            run_execution_times.append((train_time, final_eval_time, total_time))
            if log_enabled:
                progress_str = f'{output_file_name}\n{scenario}\n{accs}\n{run_execution_times}'
                with open(progress_file, 'w') as progress_write:
                    progress_write.write(progress_str)
            if device == torch.device('cuda'):
                del optimizer
                gc.collect()
                torch.cuda.empty_cache()
            print('  Run execution time: train: %.3f (s) - eval: %.3f (s)- total: %.3f (s)'%\
                  (train_time, final_eval_time, total_time))
        accs = np.array(accs)
        train_times = np.array(train_times)
        evaluation_times = np.array(evaluation_times)
        total_times = np.array(total_times)
        scenario_description = 'Scenario %d/%d - Epochs: %d - lr: - %s - dropout: %s - Weight_decay: %s - Grad_clip
        (scenario, scenario_count, epochs, lr, drop_out, weight_decay, grad_clip)
        accuracy_description = '\n  Final acc over %d instances: %.2f +- %.2f%%\n'%(runs, accs.mean(), accs.std())
```

```
                        # print(train_times.mean(), evaluation_times.mean(), total_times.mean())
                        display_str = '  %s'%(scenario_description) +\
                        '\n  Avg execution time: train: %.3f +- %.3f (s) - eval: %.3f +- %.3f (s) - total: %.3f +- %.3f (s) on %s'%
                        (train_times.mean(), train_times.std(), evaluation_times.mean(), evaluation_times.std(),
                             total_times.mean(), total_times.std(), device_name) + accuracy_description

                        #progress_str = f'{output_file_name}\n{scenario}\n{accs}'
                        print(display_str)
                        plot_str = display_str # scenario_description + accuracy_description
                        if search_plot:
                            plot_accs(accs_accross_runs_plot, plot_str, comment, save_image)
                        if log_enabled:
                            mode = 'a' if os.path.exists(output_file_path) else 'w'

                            with open(output_file_path, mode) as output_write:
                                output_write.write(display_str)
                        ran_in_middle = False
                        next_run = 1
                        scenario += 1
```

```
Running on Tesla P100-PCIE-16GB
Using CIFAR-10 test set
FINAL EVAL - ResNet9 + Cosine Loss + modified train transforms + random search best config
Files already downloaded and verified
Files already downloaded and verified

Scenario 1/1 - Epochs: 700 - lr: - 0.0001 - dropout: 0 - Weight_decay: 0.00016051911333587627 - Grad clip: 0.02576638574613588
  Run# [1/5] - Num Samples For Training 100 - Num Samples For Val 2000
   Train Epoch: 1 [step 1/1 (0%)]       Loss: 0.993465
   Train Epoch: 101 [step 1/1 (0%)]     Loss: 0.382546
   Train Epoch: 201 [step 1/1 (0%)]     Loss: 0.200414
   Train Epoch: 301 [step 1/1 (0%)]     Loss: 0.074535
   Train Epoch: 401 [step 1/1 (0%)]     Loss: 0.034683
   Train Epoch: 501 [step 1/1 (0%)]     Loss: 0.015287
   Train Epoch: 601 [step 1/1 (0%)]     Loss: 0.016038
   Train Epoch: 700 [step 1/1 (0%)]     Loss: 0.015774
   Test set: Average loss: 0.0044, Accuracy: 729/2000 (36.45%)
  Run execution time: train: 155.332 (s) - eval: 0.764 (s)- total: 156.096 (s)
  Run# [2/5] - Num Samples For Training 100 - Num Samples For Val 2000
   Train Epoch: 1 [step 1/1 (0%)]       Loss: 1.054169
   Train Epoch: 101 [step 1/1 (0%)]     Loss: 0.376461
   Train Epoch: 201 [step 1/1 (0%)]     Loss: 0.170049
   Train Epoch: 301 [step 1/1 (0%)]     Loss: 0.074577
   Train Epoch: 401 [step 1/1 (0%)]     Loss: 0.035534
   Train Epoch: 501 [step 1/1 (0%)]     Loss: 0.021809
   Train Epoch: 601 [step 1/1 (0%)]     Loss: 0.016703
   Train Epoch: 700 [step 1/1 (0%)]     Loss: 0.009904
   Test set: Average loss: 0.0042, Accuracy: 759/2000 (37.95%)
  Run execution time: train: 154.445 (s) - eval: 0.759 (s)- total: 155.204 (s)
  Run# [3/5] - Num Samples For Training 100 - Num Samples For Val 2000
```

```
   Train Epoch: 1 [step 1/1 (0%)]       Loss: 0.936893
   Train Epoch: 101 [step 1/1 (0%)]     Loss: 0.373526
   Train Epoch: 201 [step 1/1 (0%)]     Loss: 0.174713
   Train Epoch: 301 [step 1/1 (0%)]     Loss: 0.063958
   Train Epoch: 401 [step 1/1 (0%)]     Loss: 0.034751
   Train Epoch: 501 [step 1/1 (0%)]     Loss: 0.017030
   Train Epoch: 601 [step 1/1 (0%)]     Loss: 0.012998
   Train Epoch: 700 [step 1/1 (0%)]     Loss: 0.012045
   Test set: Average loss: 0.0043, Accuracy: 780/2000 (39.00%)
  Run execution time: train: 154.273 (s) - eval: 0.771 (s)- total: 155.044 (s)
  Run# [4/5] - Num Samples For Training 100 - Num Samples For Val 2000
   Train Epoch: 1 [step 1/1 (0%)]       Loss: 1.045752
   Train Epoch: 101 [step 1/1 (0%)]     Loss: 0.339778
   Train Epoch: 201 [step 1/1 (0%)]     Loss: 0.170568
   Train Epoch: 301 [step 1/1 (0%)]     Loss: 0.077435
   Train Epoch: 401 [step 1/1 (0%)]     Loss: 0.031442
   Train Epoch: 501 [step 1/1 (0%)]     Loss: 0.019418
   Train Epoch: 601 [step 1/1 (0%)]     Loss: 0.010596
   Train Epoch: 700 [step 1/1 (0%)]     Loss: 0.006670
   Test set: Average loss: 0.0039, Accuracy: 886/2000 (44.30%)
  Run execution time: train: 153.679 (s) - eval: 0.752 (s)- total: 154.431 (s)
  Run# [5/5] - Num Samples For Training 100 - Num Samples For Val 2000
   Train Epoch: 1 [step 1/1 (0%)]       Loss: 0.909625
   Train Epoch: 101 [step 1/1 (0%)]     Loss: 0.336025
   Train Epoch: 201 [step 1/1 (0%)]     Loss: 0.190095
   Train Epoch: 301 [step 1/1 (0%)]     Loss: 0.072564
   Train Epoch: 401 [step 1/1 (0%)]     Loss: 0.036186
   Train Epoch: 501 [step 1/1 (0%)]     Loss: 0.022619
   Train Epoch: 601 [step 1/1 (0%)]     Loss: 0.010899
   Train Epoch: 700 [step 1/1 (0%)]     Loss: 0.018374
   Test set: Average loss: 0.0045, Accuracy: 734/2000 (36.70%)
  Run execution time: train: 153.257 (s) - eval: 0.752 (s)- total: 154.009 (s)
  Scenario 1/1 - Epochs: 700 - lr: - 0.0001 - dropout: 0 - Weight_decay: 0.00016051911333587627 - Grad_clip: 0.02576638574613588
  Avg execution time: train: 154.197 +- 0.708 (s) - eval: 0.760 +- 0.007 (s) - total: 154.957 +- 0.713 (s) on Tesla P100-PCIE-16GB
  Final acc over 5 instances: 38.88 +- 2.86%

CPU times: user 11min 24s, sys: 1min 32s, total: 12min 56s
Wall time: 12min 56s
```

COMP 691 - Deep Learning project - Challenge 1

Team: DL_DT_Explore

Members:

- Trong-Tuan Tran
- Hussein Abdallah
- Manh-Quoc-Dat Le

Jupyter Notebook implement the approach: ResNet9 model for feature embedding using triplet loss and SVM classifier.

Here the goal is to train on 100 samples. In this preliminary testbed the evaluation will be done on a 2000 sample validation set. Note in the end the final evaluation will be done on the full CIFAR-10 test set as well as potentially a separate dataset. The validation samples here should not be used for training in any way, the final evaluation will provide only random samples of 100 from a datasource that is not the CIFAR-10 training data.

Initial configurations:

(Please note that due to the need to support resume random search capability, some hyperparameters will need to be defined later, after the environment detecting section)

In [8]:

```
import time
import pickle
from scipy.stats import loguniform
from numpy.random import RandomState
import torchvision
import numpy as np
import torch
import torch.optim as optim
from torch.utils.data import Subset
from torchvision import datasets, transforms

search_plot = True
log_enabled = False
save_image = True
summarize = False
plot_distance = True
google_drive_mount = True # If running in Google Colab and enable this, progress and image files will save to Google Drive instead o
final_eval = True # Whether to run using train set or test set for final evalutation & submission
linear_classifier = 'SVM' #'KNN' # Decide which linear classifier to use on top of the ResNet9 layers, on the embed features
batch_multiplier = 10 # Number of augmented minibatches to be accumulated into a big batch to generate online triplets
hard_neg_prob = torch.tensor(0.1) # Probability of a accumulated batch to be mined with hard negative or semi-hard negative triplet
```

```
batch_size = 500
runs = 3 # Number of instances to run the train and test to evaluate mean and std dev of test accuracy

epoch_display_range = 20

eval_str = 'DEV PHASE - ' if not final_eval else 'FINAL EVAL - '

# Comments to put on accuracy curve plot:
comment = eval_str + f'ResNet9 + Data Augmentation + no lr sched + Triplet loss ({(hard_neg_prob*100):.1f}% hard negative) + SVM'
if final_eval:
    search_plot = False
    save_image = False
    google_drive_mount = False
    log_enabled = False

use_cuda = torch.cuda.is_available()
device = torch.device("cuda" if use_cuda else "cpu")
#device = torch.device('cpu')
```

The section below detects which environment is running (Colab, Kaggle or local computer). The output folders will be determine accordingly. If running in final_eval mode, no output file will be generated.

In [9]:

```
import os
output_path = 'output_txt/'
img_path = 'img/'
Colab = False
Kaggle = 'kaggle' in os.getcwd()
root = '.' # Root to download dataset
if 'google.colab' in str(get_ipython()):
    print('Running on CoLab')
    Colab = True
    from google.colab import drive
    if not os.path.exists('/content/drive/MyDrive/') and google_drive_mount:
        drive.mount('/content/drive', force_remount=False)

    else:
        if google_drive_mount:
            print('Drive already mounted at at /content/drive')

    Google_path = '/content/drive/MyDrive/Colab Notebooks/COMP691_project/' if google_drive_mount else '/'
    if not os.path.exists(Google_path):
        os.mkdir(Google_path)
    img_path = Google_path + img_path
    if not os.path.exists(img_path):
        os.mkdir(img_path)
    output_path = Google_path + output_path
else:
```

```
    if Kaggle:
        root = '../input/cifar10'
        output_path = '/'
        img_path = 'img/'
        print('Running in Kaggle')
    else:
        print('Not running on CoLab or Kaggle')

output_file_name = 'report_ADAM_Triplet_SVM_ResNet9_GridSearch_FINAL.txt'
output_file_path = output_path + output_file_name
progress_file = output_path + 'Triplet_SVM_grid_search_ResNet9_progress_FINAL.txt'
img_file_name_prefix = output_file_name.replace('.txt', '')
img_file_path = img_path + img_file_name_prefix + '/'
save_state_file_path = output_file_path.replace('.txt', '.pkl')

if not final_eval:
    if not os.path.exists(img_path):
        os.mkdir(img_path)

    if not os.path.exists(img_file_path):
        os.mkdir(img_file_path)

    if not os.path.exists(output_path):
        os.mkdir(output_path)
```

```
Running on CoLab
```

This section define hyperparameters and check if there is an pending run not completed before. If yes, it will resume using the stored settings regardless of the entered hyperparameters below.

In [10]:

```
save_state = {}
if summarize:
    if not log_enabled:
        raise NameError('log_enabled should be True in order to enable summarize!')
    if os.path.exists(save_state_file_path):
        with open(save_state_file_path ,'rb') as dataHandle:
            save_state = pickle.load(dataHandle)

accumulated_accs = []
# Epochs: 300 - lr: - 0.001 - dropout: 0 - Weight_decay: 1e-05 - Grad_clip: 0.005
# Epochs: 700 - lr: - 0.0001 - dropout: 0 - Weight_decay: 0.00016052 - Grad_clip: 0.01512
if len(save_state) == 0:
    epochs_list = [400]
    save_state['epochs'] =  epochs_list

    # grad_clips = sorted(list(loguniform(1e-4, 1).rvs(5, random_state=0)))
    grad_clips = [0.005] # Gradient clipping
```

```
    save_state['grad_clips'] = grad_clips
    # weight_decays = sorted(list(loguniform(1e-5, 1e-3).rvs(5, random_state=0)))
    weight_decays = [1e-5] # Weight decay for Adam optimizer

    save_state['weight_decays'] = weight_decays

    lrs = [0.005] # learning rates
    save_state['lrs'] = lrs

    alphas = [0.3] # margin in triplet loss
    save_state['alphas'] = alphas

    ks = [9] # k in k-Nearest Neighbor classifier
    save_state['ks'] = ks
    #drop_outs = [0, 0.1, 0.2]
    drop_outs = [0]
    save_state['drop_outs'] = drop_outs
    if summarize:
        with open(save_state_file_path, 'wb') as dataHandle:
            pickle.dump(save_state, dataHandle)

    print(f'First time run on profile {output_file_name}')
else:
    epochs_list = save_state['epochs']
    grad_clips = save_state['grad_clips']
    weight_decays = save_state['weight_decays']
    lrs = save_state['lrs']
    alphas = save_state['alphas']
    ks = save_state['ks']
    drop_outs = save_state['drop_outs']
    accumulated_accs = save_state['accs']
    print(f'Successfully loaded save state from profile {output_file_name}')
```

```
First time run on profile report_ADAM_Triplet_SVM_ResNet9_GridSearch_FINAL.txt
```

Setup training, testing and other helper functions.

In [11]:

```
import gc
import math
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.model_selection import RandomizedSearchCV
from matplotlib import pyplot as plt
import numpy as np
from numpy import unravel_index
import torch
```

```
def train(model, device, train_loader, optimizer, epoch, alpha=0.2, grad_clip=None,
          sched=None, display=True, distance_visualize=True, scenario_description='', svm=False):

    loss_function = nn.TripletMarginLoss(margin=alpha)
    model = model.to(device)
    acc_data = torch.tensor([])
    acc_target = torch.tensor([])
    for i in range(batch_multiplier):
        for data, target in train_loader:
            acc_data = torch.cat([acc_data, data])
            acc_target = torch.cat([acc_target, target])


    #print(acc_data.size(0))
    hard_negative = bool(torch.bernoulli(hard_neg_prob).item())
    if not hard_negative:
    #data, target = data.to(device), target.to(device)
        (data_a, data_p, data_n), _ = triplet_generate_random((acc_data, acc_target), timing=False)
    else:
        (data_a, data_p, data_n), _ = triplet_generate_hard_negative_mining((acc_data, acc_target), model, alpha, device, timing=Fa
    l = 0
    for i in range(0, data_a.size(0), batch_size):
        data_a_batch = data_a[i:i+batch_size].to(device)
        data_p_batch = data_p[i:i+batch_size].to(device)
        data_n_batch = data_n[i:i+batch_size].to(device)

        optimizer.zero_grad()
        #data_a, data_p, data_n = data_a.to(device), data_p.to(device), data_n.to(device)
        model.train()
        embedded_a = model(data_a_batch)
        embedded_p = model(data_p_batch)
        embedded_n = model(data_n_batch)

        loss = loss_function(embedded_a, embedded_p, embedded_n)
        #loss = F.cross_entropy(output, target)
        loss.backward()
        l += loss.detach().item()
        if grad_clip:
            nn.utils.clip_grad_value_(model.parameters(), grad_clip)
        optimizer.step()
        if sched:
            sched.step()
        del loss, embedded_a, embedded_p, embedded_n, data_a_batch, data_p_batch, data_n_batch
        gc.collect()
        if device == torch.device('cuda'):
            torch.cuda.empty_cache()
```

```
    if svm:
        with torch.no_grad():
            embeded_data = model(acc_data.to(device))
        print('  Start fitting svm model with random search cv...')
        model.fit(embeded_data, acc_target)
    if display:
        print('   Train Epoch: {}\tAvg Loss: {:.6f}'.format(
            epoch + 1, l/int(data_a.size(0)/batch_size)))
    embeded_data = None
    if distance_visualize or not svm:
        model.eval() # switch to eval mode to store a copy of current embed data to be used for k-NN predict in test phase
        with torch.no_grad():
            embeded_data = model(acc_data.to(device))
            embed_dict = {'embed_data': (embeded_data, acc_target)}
            embed_data_file = 'embed_data.pkl'
            with open(embed_data_file, 'wb') as dataHandle:
                pickle.dump(embed_dict, dataHandle)
            #model.embeded_data =
        if distance_visualize:
            comment = f'Epoch: {epoch + 1} - Batch multiplier: x{batch_multiplier} - hard negative prob: {hard_neg_prob.item():.2f}
            visualize_distances((embeded_data, acc_target), display_str=scenario_description, comment=comment, save_img=save_image)

    del acc_data, acc_target, embeded_data, embed_dict
    gc.collect()
    if device == torch.device('cuda'):
        torch.cuda.empty_cache()

def test(model, device, test_loader, k, display=True, distance_visualize=True, svm=False):
    model.eval()
    model = model.to(device)
    test_loss = 0
    correct = 0
    correct_knn = 0
    #loss_function = nn.CrossEntropyLoss()
    classifier = 'k-NN'
    benchmark_str = ''
    data_embedded = None
    if distance_visualize or not svm:
        embed_data_file = 'embed_data.pkl'
        if os.path.exists(embed_data_file):
            with open(embed_data_file ,'rb') as dataHandle:
                embed_data_dict = pickle.load(dataHandle)
        train_embeded_data, labels = embed_data_dict['embed_data']
        train_embeded_data, labels = train_embeded_data.to(device), labels.to(device)

    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
```

```
            data_embedded = None
            if not svm:
                data_embedded = model(data)
                output = k_nearest_neighbors(data_embedded, (train_embeded_data, labels), k).to(device)
                #print(output.shape, target.shape)
                #test_loss += loss_function(output, target).item() # sum up batch loss
                #test_loss += F.cross_entropy(output, target, size_average=False).item()
                #pred = output.max(1, keepdim=True)[1] # get the index of the max log-probability

            else:
                classifier = 'SVM'
                output = model(data, svm_predict=svm).to(device)
                if distance_visualize:
                    data_embedded = model(data)
                    output_knn = k_nearest_neighbors(data_embedded, (train_embeded_data, labels), k).to(device)

            correct += output.eq(target.view_as(output)).sum().item()
            if svm & distance_visualize:
                correct_knn += output_knn.eq(target.view_as(output)).sum().item()
                benchmark_str = f' - k-NN acc benchmark: {correct_knn}/{len(test_loader.dataset)} ({100. * correct_knn / len(test_l
            del output, data, target, data_embedded
            gc.collect()
            if device == torch.device('cuda'):
                torch.cuda.empty_cache()
    test_loss /= len(test_loader.dataset)
    if display:
        print('   Test set: Accuracy: {}/{} ({:.2f}%) - classifier: {}'.format(
            correct, len(test_loader.dataset),
            100. * correct / len(test_loader.dataset), classifier) + benchmark_str)

    del train_embeded_data, labels, embed_data_dict
    gc.collect()
    if device == torch.device('cuda'):
        torch.cuda.empty_cache()
    return 100. * correct / len(test_loader.dataset)


def triplet_generate_random(minibatch, timing=False):
    start_time = time.time()
    image, labels = minibatch
    label_set = labels.unique().tolist()
    label_to_indices = {label: np.where(labels.numpy() == label)[0] for label in label_set}
    #print(label_to_indices)
    idx_pos = []
    idx_neg = []

    for idx, label in enumerate(labels):
        positive_index = idx
```

```
        #print(positive_index, idx)
        while positive_index == idx:
            positive_index = np.random.choice(label_to_indices[label.item()])
            #print(positive_index)
        negative_labels = np.random.choice(list(set(label_set) - set([label.item()])))
        negative_index = np.random.choice(label_to_indices[negative_labels])
        idx_pos.append(positive_index)
        idx_neg.append(negative_index)
    if timing:
        execution_time = time.time() - start_time
        print(f'Found [{len(idx_pos)}/{labels.size(0)}] random triplets - Duration: {execution_time:.3f}s')
    return (image, image[idx_pos], image[idx_neg]), (labels, labels[idx_pos], labels[idx_neg])


def triplet_generate_hard_negative_mining(minibatch, model, margin, device, timing=False):
    start_time = time.time()
    image, labels = minibatch
    image, labels = image.to(device), labels.cpu()
    model = model .to(device)
    model.train()
    semi_hard_prob = torch.tensor(0.4)

    semi_hard_count = 0
    label_set = labels.unique().tolist()
    label_to_indices = {label: np.where(labels.numpy() == label)[0] for label in label_set}


    idx_anchor = []
    idx_pos = []
    idx_neg = []

    with torch.no_grad():
        embed_data = model(image)
    #print(embed_data.max(), embed_data.min(), embed_data.mean() ,embed_data.std())
    distances = torch.cdist(embed_data, embed_data)
    sorted_distances, sort_indices = distances.topk(k=distances.size(1), largest=False)
    missed_hard_neg_count = 0
    missed_semi_hard_neg_count = 0
    for i, label in enumerate(labels):
        #print(f'label_idx: {i} - class: {label.item()}')
        triplet_selected = False
        current_sort_distances = sorted_distances[i]
        current_sort_indices = sort_indices[i]
        semi_hard = bool(torch.bernoulli(semi_hard_prob).item())
        pos_index, neg_index, sh_count, semi_hard = triplet_pair_select_hard_negative(i, current_sort_distances, current_sort_indic
                                                                                      labels, margin, semi_hard)

        if pos_index != -1 and neg_index != -1:
```

```
            idx_anchor.append(i)
            idx_pos.append(pos_index)
            idx_neg.append(neg_index)
            if semi_hard:
                semi_hard_count += sh_count
        else:
            if semi_hard:
                missed_semi_hard_neg_count += 1
            else:
                missed_hard_neg_count += 1
    del embed_data
    gc.collect()
    if device == torch.device('cuda'):
        torch.cuda.empty_cache()
    if timing:
        execution_time = time.time() - start_time
        print(f'   Found [{len(idx_anchor)}/{labels.size(0)}] pairs of hard-negative triplets in current minibatch, semi-hard pairs
              f'- Missed hard pair(s): {missed_hard_neg_count} - Missed semi-hard pair(s): {missed_semi_hard_neg_count} - Duration:
    return (image[idx_anchor], image[idx_pos], image[idx_neg]), (labels[idx_anchor], labels[idx_pos], labels[idx_neg])

def triplet_pair_select_hard_negative(idx_anchor, current_sort_distances, current_sort_indices, labels, margin, semi_hard=False):
    idx_pos = -1
    idx_neg = -1
    label_a = labels[idx_anchor].item()
    triplet_pair_found = False
    semi_hard_count = 0
    #print('semi-hard:', semi_hard)

    for j, candidate_distance_n in enumerate(current_sort_distances): # Start from the left (smallest distances)

        if j == idx_anchor:
            continue # skip if current column is the anchor itself

        distance_idx_j = current_sort_indices[j].item()
        candidate_label_n = labels[distance_idx_j].item()
        #print(f'j: {j} - candidate_distance_n: {candidate_distance_n}, candidate_label_n: {candidate_label_n}')
        if candidate_label_n == label_a:
            continue
        k = len(current_sort_distances) - 1
        while not triplet_pair_found and k >= 0:
            if k == idx_anchor:
                k -= 1
                continue
            candidate_distance_p = current_sort_distances[k]
            distance_idx_k = current_sort_indices[k].item()
            candidate_label_p = labels[distance_idx_k].item()

            if candidate_label_p == label_a:
```

```
                if not semi_hard:
                    if candidate_distance_n.item() < candidate_distance_p.item():
                        triplet_pair_found = True
                        idx_pos = distance_idx_k
                        idx_neg = distance_idx_j
                        #print(f'  k: {k} - candidate_distance_p: {candidate_distance_p}, candidate_label_p: {candidate_label_p}')
                        break
                    #else:
                        #if k == j & j == :
                            #semi_hard = True # if cannot find hard negative pair, switch to to continue searching for a semi-hard p

                else: # semi-hard mining
                    if k >= j:
                        k -= 1
                        continue
                    if candidate_distance_n.item() > candidate_distance_p.item() and \
                        candidate_distance_n.item() < candidate_distance_p.item() + margin:
                        triplet_pair_found = True
                        idx_pos = distance_idx_k
                        idx_neg = distance_idx_j
                        semi_hard_count = 1
                        #print(f'  k: {k} - candidate_distance_p: {candidate_distance_p}, candidate_label_p: {candidate_label_p}')
                        break
                    if k < j and candidate_distance_n.item() > candidate_distance_p.item() + margin:
                        break # early break moving positive to the left (while loop) if negative already goes exceeded positive + ma

                        #semi_hard = False
                        #k = len(current_sort_distances) - 1
                        #continue # if canno find semi-hard pair, reset k and switch to find searching for hard neg mining
            k -= 1

        if triplet_pair_found:
            break

    return idx_pos, idx_neg, semi_hard_count, semi_hard

def k_nearest_neighbors(minibatch_embbed, trained_embeded_set, k=1):
    minibatch_embbed = minibatch_embbed
    train_embeded_data, train_labels = trained_embeded_set
    predicts = np.zeros(minibatch_embbed.size(0))
    distances = torch.cdist(train_embeded_data, minibatch_embbed).cpu().numpy()
    #print(distances)
    k_min_indices = np.argpartition(distances, k, axis=0)[:k]
    #print(k_min_indices)
    #predicts = train_labels[distances.min(dim=0)[1]]
    for i, indices in enumerate(k_min_indices.T.tolist()):
        label_count = torch.zeros(10)
        top_k_labels = train_labels[indices]
```

```
        #print(top_k_labels)
        #print(top_k_labels.unique(return_counts=True))
        labels, counts = top_k_labels.unique(return_counts=True)
        for label, count in zip(labels.tolist(), counts.tolist()):
            label = int(label)
            label_count[label] = count
        predicts[i] = np.argmax(label_count)
    #predict = train_labels[k_min_indices]
    #print(predicts)
    return torch.from_numpy(predicts)
```

Section below define functions for plotting and performance analysis on grid search/ random search:

In [12]:

```
def generate_image_suffix():
    return f'_{time.time()%10000000:.0f}' + '.png'

def performance_summary(accumulated_accs):
    scenarios = len(accumulated_accs)
    str_output = f'\nCurrent performance summary over {scenarios} completed scenario(s): '
    top_20_percent_count = math.ceil(scenarios/5) # Top k performances
    k = top_20_percent_count
    top_20_final_accs = [(1, 0, 0) for _ in range(1, k + 1)] # (scenario, final acc means, std)
    top_20_max_accs = [(1, 0, 0, 0) for _ in range(1, k + 1)] # (scenario, run, epoch, max_acc)
    top_20_least_variant = [(1, 0, 999) for _ in range(1, k + 1)] # (scenario, final acc means, std)
    for scenario, scenario_accs in enumerate(accumulated_accs):

        scenario_accs = np.array(scenario_accs)
        run_final_accs_mean, run_final_acc_std = scenario_accs[:, -1].mean(), scenario_accs[:, -1].std()
        #print(f'scen {scenario} - acc {run_final_accs_mean:.3f} - std {run_final_acc_std:.3f}')
        idx = np.argpartition(scenario_accs, -k, axis=None)
        top_k_scenario_max_indices = [unravel_index(i, scenario_accs.shape) for i in np.sort(idx[-k:])]
        #print(top_k_scenario_max_indices)
        for run, epoch in top_k_scenario_max_indices:
            #print(run, epoch)
            top_scenario_max_acc = scenario_accs[run, epoch]
            top_20_max_accs.append((scenario, run, epoch, top_scenario_max_acc))
        #print(top_20_max_accs)
        if k < len(top_20_max_accs):
            global_max_accs = [top_scenario_max_acc for (_, _, _, top_scenario_max_acc) in top_20_max_accs]
            idx = np.argpartition(global_max_accs, -k, axis=None)
            top_20_max_accs = [top_20_max_accs[i] for i in sorted(idx[-k:].tolist(), reverse=True)]
        #for i, (max_scen, max_acc_mean, max_acc_std) in enumerate(top_20_final_accs):
        top_20_final_accs.append((scenario, run_final_accs_mean, run_final_acc_std))
        if k < len(top_20_final_accs):
            global_final_accs = [run_final_accs_mean for (_, run_final_accs_mean, _) in top_20_final_accs]
            idx = np.argpartition(global_final_accs, -k, axis=None)
            top_20_final_accs = [top_20_final_accs[i]  for i in sorted(idx[-k:].tolist(), reverse=True)]
```

```
        top_20_least_variant.append((scenario, run_final_accs_mean, run_final_acc_std))
        #print(top_20_least_variant)
        if k < len(top_20_least_variant):
            global_final_stds = [run_final_acc_std for (_, _, run_final_acc_std) in top_20_least_variant]
            #print(global_final_stds)
            idx = np.argpartition(global_final_stds, k, axis=None)
            #print(idx)
            top_20_least_variant = [top_20_least_variant[i] for i in sorted(idx[:k].tolist())]
    str_output += f'\n - Top {k} final average accuracy:'
    for (scenario, run_final_accs_mean, run_final_acc_std) in top_20_final_accs:
        str_output += f'\n   + Scenario: {scenario + 1} - Final average accuracy: {run_final_accs_mean:.2f} +- {run_final_acc_std:.
    str_output += f'\n - Top {k} max accuracy:'
    for (scenario, run, epoch, top_scenario_max_acc) in top_20_max_accs:
        str_output += f'\n   + Scenario: {scenario + 1} - run {run + 1} - epoch {epoch + 1} - Max accuracy: {top_scenario_max_acc:.
    str_output += f'\n - Top {k} least final accuracy variation:'
    for (scenario, run_final_accs_mean, run_final_acc_std) in top_20_least_variant:
        str_output += f'\n   + Scenario: {scenario + 1} - Final average accuracy: {run_final_accs_mean:.2f} +- {run_final_acc_std:.
    return str_output

def plot_accs(accs_accross_runs, display_str, comment='N/A', save_img=True):
    plt.figure();
    #epochs = list(range(1, len(accs_accross_runs[0]) + 1))
    max_acc_display = ''
    for i, run_accs in enumerate(accs_accross_runs):
        max_acc = max(run_accs)
        max_epochs = [index + 1 for index, acc in enumerate(run_accs) if acc == max_acc]
        max_acc_display = f' - max acc: {max_acc}% at epochs {max_epochs}'
        plt.plot(run_accs, label=f'Run #{i + 1}' + max_acc_display)
    plt.xlabel('epochs')
    plt.ylabel('Test accuracy (%)')
    plt.legend();
    plt.title(f'Test accuracies for \n{display_str}Note: {comment}');

    if save_img:
        scenario = display_str[display_str.index(' ') + 1: display_str.index('/')]
        img_name = img_file_path + f'scenario{scenario}_' + generate_image_suffix()
        plt.savefig(img_name, bbox_inches='tight')


def visualize_distances(minibatch, display_str='', comment='N/A', save_img=True):
    data, labels = minibatch
    class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
    label_set = labels.unique().tolist()
    label_to_indices = {label: np.where(labels.numpy() == label)[0] for label in label_set}
    data_PCA = PCA(n_components=2).fit_transform(data.cpu().numpy())
    data_TSNE = TSNE(n_components=2, n_iter=1000, learning_rate=250).fit_transform(data.cpu().numpy())
    #print(data_PCA.shape)
    plt.figure(figsize=(10, 15))
```

```
    plt.subplot(2,1,1)
    for label in label_set:
        label = int(label)
        plot_data = data_PCA[label_to_indices[label]]
        #print(plot_data.shape)
        plt.scatter(plot_data[:, 0], plot_data[:, 1], marker='+', label=f'Class {label} ({class_names[label]})')
    plt.xlabel('x1')
    plt.ylabel('x2')
    plt.legend()
    plt.title('Input minibatch\'s embed features visualization in 2D with PCA transform' +
              '\n' + display_str + '\nNote:'+ comment)
    plt.subplot(2,1,2)
    for label in label_set:
        label = int(label)
        plot_data = data_TSNE[label_to_indices[label]]
        #print(plot_data.shape)
        plt.scatter(plot_data[:, 0], plot_data[:, 1], marker='+', label=f'Class {label} ({class_names[label]})')
    plt.xlabel('x1')
    plt.ylabel('x2')
    plt.legend()
    plt.title('Input minibatch \'s embed features visualization in 2D with T-SNE' +
              '\n' + display_str + '\nNote: '+ comment)
    if save_img:
        scenario = display_str[display_str.index(' ') + 1: display_str.index('/')]
        run = display_str[display_str.index('run:')+ 5 :]
        img_name = img_file_path + f'scenario{scenario}_run{run}_distances' + generate_image_suffix()

        plt.savefig(img_name, bbox_inches='tight')
        print(f'  Save distane plot in {img_name}')
```

ResNet9 model: Use a SVC head to perform final prediction, and the SVC will be trained on the accumulated batch at the final epoch

In [13]:

```
import torch.nn as nn
import torch.nn.functional as F
from sklearn.svm import SVC

num_classes = 10
in_channels = 3

def conv_block(in_channels, out_channels, drop_out=0, pool=False):
    layers = [nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1),
              nn.BatchNorm2d(out_channels),
              nn.ReLU(inplace=True), nn.Dropout(drop_out)
              ]
    if pool: layers.append(nn.MaxPool2d(2))
    return nn.Sequential(*layers)
```

```
class NET(nn.Module):
    def __init__(self, in_channels=3, num_classes=10, drop_out=0):
#         super().__init__()
#         # Use a pretrained model
#         self.network = models.resnet34(pretrained=True)
#         # Replace last layer
#         num_ftrs = self.network.fc.in_features
#         self.network.fc = nn.Linear(num_ftrs, num_classes)
        super().__init__()

        self.conv1 = conv_block(in_channels, 64, drop_out)
        self.conv2 = conv_block(64, 128, drop_out, pool=True)
        self.res1 = nn.Sequential(conv_block(128, 128, drop_out), conv_block(128, 128, drop_out))
        self.dropout = nn.Dropout(drop_out)
        self.conv3 = conv_block(128, 256, pool=True)
        self.conv4 = conv_block(256, 512, pool=True)
        self.res2 = nn.Sequential(conv_block(512, 512, drop_out), conv_block(512, 512, drop_out))
        self.conv5 = conv_block(512, 1028, drop_out, pool=True)
        self.res3 = nn.Sequential(conv_block(1028, 1028, drop_out), conv_block(1028, 1028, drop_out))

        self.embedding = nn.Sequential(nn.MaxPool2d(2),
                                       nn.Flatten()#, nn.Linear(1028, 512)
                                       )

        self.svm = SVC()
        self.best_params = None

    def fit(self, x, y):
        x = x.view(x.size(0), -1).cpu().numpy()
        #print(x.shape)
        y = y.cpu().numpy()
        cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
        # define search space
        space = dict()
        space['kernel'] = ['poly', 'rbf', 'sigmoid']
        space['gamma'] = ['scale', 'auto']
        space['C'] = loguniform(1e-4, 1e2).rvs(60, random_state=0)
        # define search
        search = RandomizedSearchCV(model.svm, space, n_iter=30, scoring='accuracy', n_jobs=-1, cv=cv, random_state=1)
        # execute search
        start_time = time.time()
        result = search.fit(x, y)
        # summarize result
        self.best_params = result.best_params_
        #print('  Best Score: %s' % result.best_score_)
        svm_train_time = time.time() - start_time
        print('  SVC training done in %.4f(s). Best Hyperparameters: %s' % (svm_train_time, self.best_params))
        self.svm = result.best_estimator_
```

```
        if os.path.exists(output_file_path):
            string_to_write = f'SVC best params: {result.best_params_}'
            with open(output_file_path, 'a') as f:
                f.write('\n' + string_to_write)
        #print(self.svm.get_params())
        #self.svm.fit(x, y)

    def forward(self, xb, svm_predict=False):
        out = self.conv1(xb)
        out = self.conv2(out)
        out = self.res1(out) + out
        out = self.conv3(out)
        out = self.dropout(out)
        out = self.conv4(out)
        out = self.dropout(out)
        out = self.res2(out) + out
        out = self.conv5(out)
        out = self.res3(out) + out
        out = self.embedding(out)

        if svm_predict:
            #print('Predict using svm...')
            out = self.svm.predict(out.cpu().numpy())
            out = torch.tensor(out)
        return out
```

The below tries a number of random problem instances. The number of instance to run is defined in the `run` variable at the begining.

In [14]:

```
#%%time

device_name = torch.cuda.get_device_name(0) if device == torch.device('cuda') else 'cpu'
scenario_count = len(epochs_list) * len(weight_decays) * len(lrs) * len(drop_outs) * len(grad_clips)
print(f'Total scenarios: {scenario_count}')
for key, value in save_state.items():
    if key != 'accs':
        print(f'{key}: {value}')

normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                 std=[0.229, 0.224, 0.225])

transform_train = transforms.Compose([
                                    transforms.RandomCrop(32, padding=4, padding_mode='reflect'),
                                    transforms.RandomGrayscale(),
                                    transforms.RandomHorizontalFlip(),
                                    torchvision.transforms.RandomAffine(degrees=30),
                                    transforms.ColorJitter(brightness=0.4, contrast=0.4, saturation=0.4, hue=0.2),
                                    #transforms.ColorJitter(),
```

```
                                      transforms.ToTensor(),
                                      normalize]) #careful to keep this one same
transform_val = transforms.Compose([transforms.ToTensor(), normalize])


print('Running on {}'.format(device_name))

##### Cifar Data
cifar_data = datasets.CIFAR10(root='.',train=True, transform=transform_train, download=True)

#We need two copies of this due to weird dataset api
cifar_data_val = datasets.CIFAR10(root='.',train=True, transform=transform_val, download=True)


# Extract a subset of 100 (class balanced) samples per class
training_done = False
count = 1
scenario = 1
next_run = 1
previous_runs_accs = []
previous_train_times = []
previous_eval_times = []
previous_exec_times = []

ran_in_middle = False

if log_enabled:
    if os.path.exists(progress_file):
        with open(progress_file, 'r') as file_read:
            progress_content = file_read.readlines()
        # print(progress_content)
        if progress_content[0].replace('\n', '') == output_file_name:
            previous_scenario = progress_content[1].replace('\n', '')
            previous_runs_accs = eval(progress_content[2].replace('\n', ''))
            previous_runs = len(previous_runs_accs)
            print(f'Previous progress on {output_file_name} stopped at scenario {previous_scenario}/{scenario_count}' +\
                 f', run {previous_runs}/{runs}')
            if previous_runs >= runs: # Already complete the previous scenario
                scenario = int(previous_scenario) + 1

            else: # Previous scenario just completed partially, resume in the next run
                ran_in_middle = True
                scenario = int(previous_scenario)
                next_run = previous_runs + 1
                previous_execution_times = eval(progress_content[3].replace('\n', ''))
                prev_accs_accross_runs_plot = eval(progress_content[4].replace('\n', ''))
                for i, previous_execution_time in enumerate(previous_execution_times):
                    previous_train_times.append(previous_execution_time[0])
```

```
                    previous_eval_times.append(previous_execution_time[1])
                    previous_exec_times.append(previous_execution_time[2])
            if scenario > scenario_count:
                training_done = True
                print('Training was already done!')
            else:
                print(f'Will resume training at scenario: {scenario}, run# {next_run}')


if not training_done:
    for epochs in epochs_list:
        for lr in lrs:
            for drop_out in drop_outs:
                for weight_decay in weight_decays:
                    for grad_clip in grad_clips:
                        for alpha in alphas:
                            for k in ks:
                                if not ran_in_middle:

                                    accs = []
                                    train_times = []
                                    evaluation_times = []
                                    total_times = []
                                    run_execution_times = []
                                    accs_accross_runs_plot = []
                                else:
                                    if count < scenario:
                                        count += 1
                                        continue #skip until reaching the scenario to run
                                    accs = previous_runs_accs
                                    train_times = previous_train_times
                                    evaluation_times = previous_eval_times
                                    total_times = previous_exec_times
                                    run_execution_times = previous_execution_times
                                    accs_accross_runs_plot = prev_accs_accross_runs_plot
                                #scenario += 1
                                scenario_description = 'Scenario %d/%d - Epochs: %d - lr: - %s - dropout: %s - Weight_decay: %s - Gr
                                    (scenario, scenario_count, epochs, lr, drop_out, weight_decay, grad_clip, alpha, k)
                                print(f'\n{scenario_description}')

                                for seed in range(next_run, runs + 1):
                                    start_time = time.time()

                                    prng = RandomState(seed)
                                    random_permute = prng.permutation(np.arange(0, 5000))
                                    indx_train = np.concatenate([np.where(np.array(cifar_data.targets) == classe)[0][random_permute
                                    indx_val = np.concatenate([np.where(np.array(cifar_data_val.targets) == classe)[0][random_permu
```

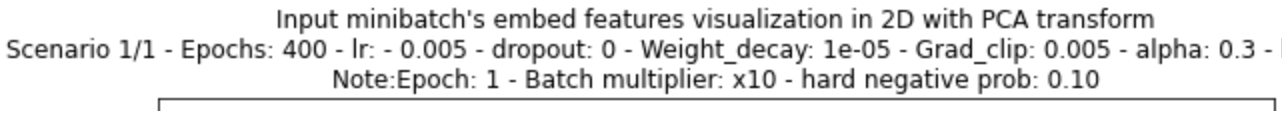
```
                train_data = Subset(cifar_data, indx_train)
                val_data = Subset(cifar_data_val, indx_val)

                print('  Run# [%d/%d] - Num Samples For Training %d - Num Samples For Val %d'%(seed, runs, trai

                train_loader = torch.utils.data.DataLoader(train_data,
                                                           batch_size=batch_size,
                                                           shuffle=True)

                val_loader = torch.utils.data.DataLoader(val_data,
                                                         batch_size=batch_size,
                                                         shuffle=False)

                model = NET(in_channels, num_classes, drop_out)
                model.to(device)
                optimizer = torch.optim.Adam(model.parameters(),
                                             lr=lr,
                                             #momentum=0.9,
                                             weight_decay=weight_decay)
                sched = torch.optim.lr_scheduler.OneCycleLR(optimizer, lr, epochs=epochs,
                        steps_per_epoch=int(100*batch_multiplier/batch_size))
                test_accs = []
                eval_time = 0
                for epoch in range(epochs):
                    print_condition = epoch%epoch_display_range == 0 or epoch== epochs - 1
                    distance_visualize = plot_distance and (epoch == 0 or epoch == epochs - 1)
                    predict_svm = linear_classifier == 'SVM' and epoch == epochs - 1
                    train(model, device, train_loader, optimizer, epoch, alpha=alpha, grad_clip=grad_clip,
                        sched=sched, display=print_condition, distance_visualize=distance_visualize,
                        scenario_description=scenario_description + f' - run: {seed}', svm=predict_svm)

                    if search_plot:
                        eval_start = time.time()
                        test_acc = test(model, device, val_loader, k, display=print_condition, distance_visuali
                        eval_time = time.time() - eval_start
                        test_accs.append(test_acc)

                train_time = time.time() - start_time
                train_times.append(train_time)
                final_eval_start = time.time()
                final_acc = test(model, device, val_loader, k, display=print_condition, distance_visualize=dist
                    if not search_plot else test_accs[-1]
                accs.append(final_acc)
                final_eval_time = time.time() - final_eval_start  if not search_plot else eval_time
                evaluation_times.append(final_eval_time)
                if search_plot:
                    accs_accross_runs_plot.append(test_accs)
```

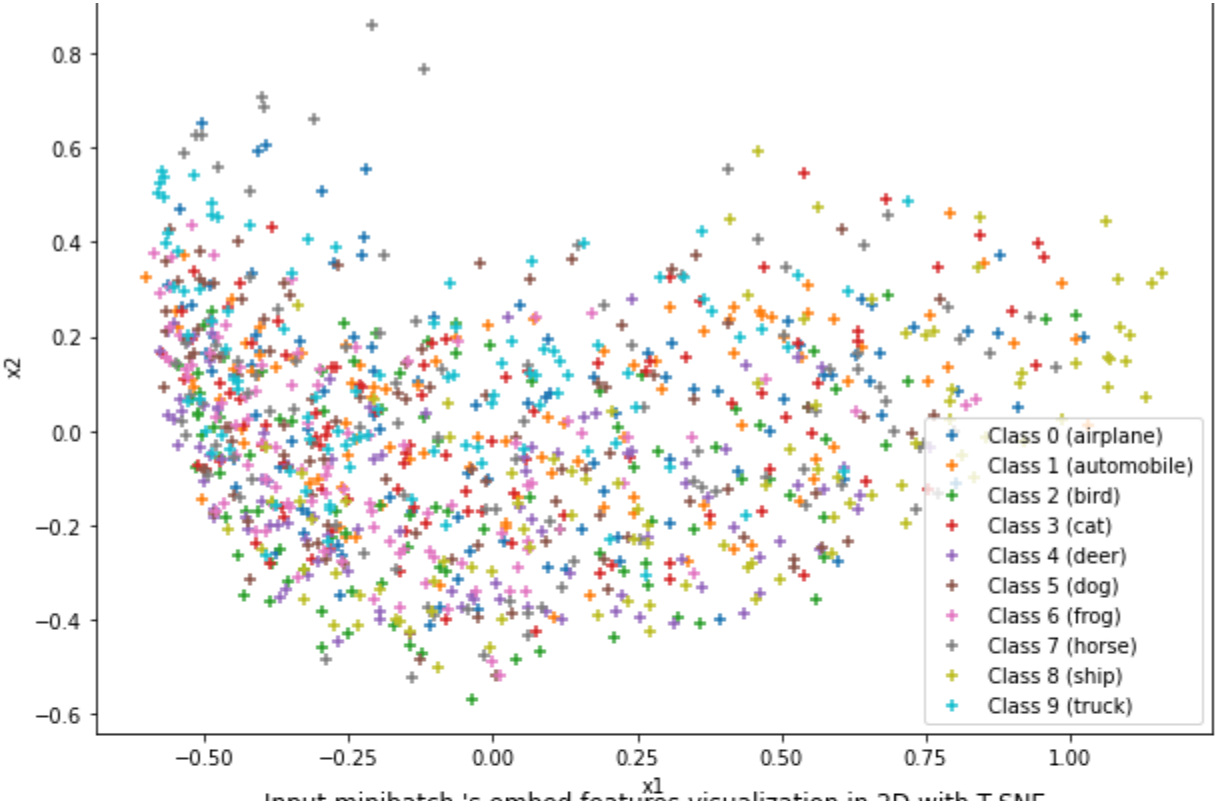
```
            total_time = time.time() - start_time
            total_times.append(total_time)
            run_execution_times.append((train_time, final_eval_time, total_time))
            if log_enabled:
                progress_str = f'{output_file_name}\n{scenario}\n{accs}\n{run_execution_times}' +\
                f'\n{accs_accross_runs_plot}'

                with open(progress_file, 'w') as progress_write:
                    progress_write.write(progress_str)

            del optimizer, model
            gc.collect()
            if device == torch.device('cuda'):

                torch.cuda.empty_cache()
            print('  Run execution time: train: %.3f (s) - eval: %.3f (s)- total: %.3f (s)'%\
                (train_time, final_eval_time, total_time))
        accs = np.array(accs)
        train_times = np.array(train_times)
        evaluation_times = np.array(evaluation_times)
        total_times = np.array(total_times)

        accuracy_description = '\n  Final acc over %d instances: %.2f +- %.2f\n'%(runs, accs.mean(), accs.s
        # print(train_times.mean(), evaluation_times.mean(), total_times.mean())
        display_str = '  %s'%(scenario_description) +\
        '\n  Avg execution time: train: %.3f +- %.3f (s) - eval: %.3f +- %.3f (s) - total: %.3f +- %.3f (s)
        (train_times.mean(), train_times.std(), evaluation_times.mean(), evaluation_times.std(),
            total_times.mean(), total_times.std(), device_name) + accuracy_description
        accumulated_accs.append(accs_accross_runs_plot)
        #progress_str = f'{output_file_name}\n{scenario}\n{accs}'
        print(display_str)
        if search_plot:
            plot_str = display_str # scenario_description + accuracy_description
            plot_accs(accs_accross_runs_plot, plot_str, comment, save_image)
        if summarize:
            save_state['accs'] = accumulated_accs
            with open(save_state_file_path, 'wb') as dataHandle:
                pickle.dump(save_state, dataHandle)
            summary = performance_summary(accumulated_accs)
            print(summary)
            display_str += summary

        if log_enabled:
            mode = 'a' if os.path.exists(output_file_path) else 'w'

            with open(output_file_path, mode) as output_write:
                output_write.write(display_str)
        ran_in_middle = False
```

```
                                    next_run = 1
                                    scenario += 1
```

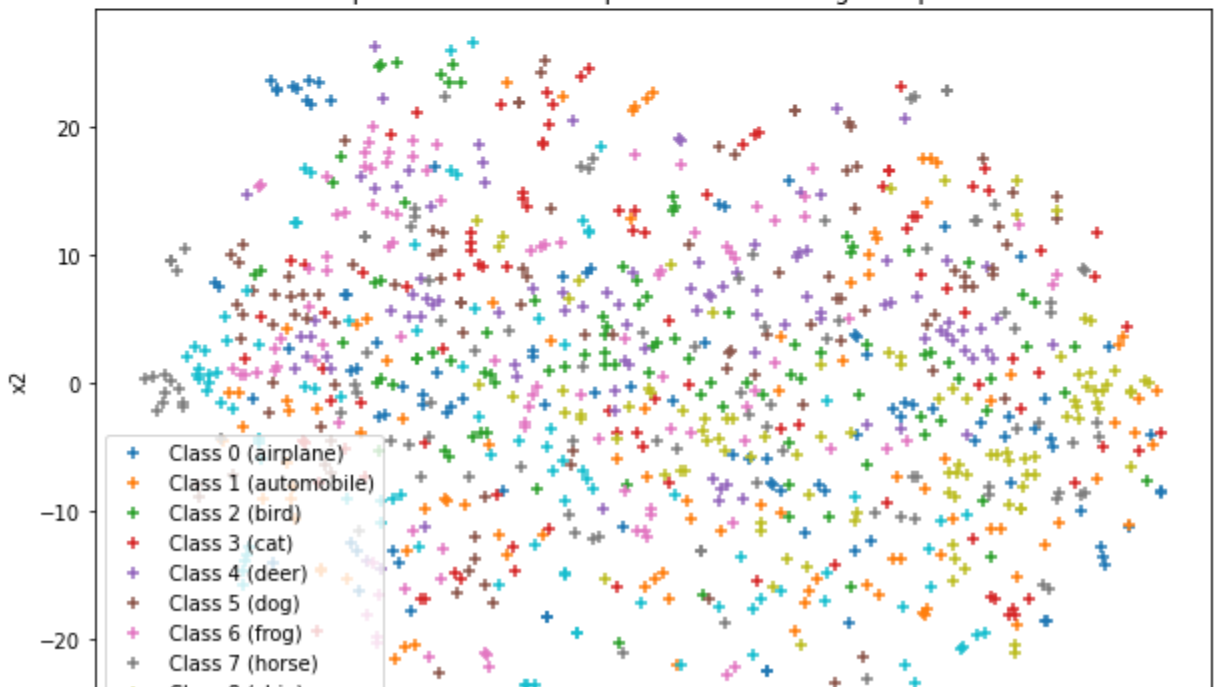
```
Total scenarios: 1
epochs: [400]
grad_clips: [0.005]
weight_decays: [1e-05]
lrs: [0.005]
alphas: [0.3]
ks: [9]
drop_outs: [0]
Running on Tesla P100-PCIE-16GB
Files already downloaded and verified
Files already downloaded and verified

Scenario 1/1 - Epochs: 400 - lr: - 0.005 - dropout: 0 - Weight_decay: 1e-05 - Grad_clip: 0.005 - alpha: 0.3 - k=9
  Run# [1/3] - Num Samples For Training 100 - Num Samples For Val 2000
   Train Epoch: 1       Avg Loss: 11.709613
   Train Epoch: 21      Avg Loss: 0.233154
   Train Epoch: 41      Avg Loss: 0.224730
   Train Epoch: 61      Avg Loss: 0.225647
   Train Epoch: 81      Avg Loss: 0.225557
   Train Epoch: 101     Avg Loss: 0.194813
   Train Epoch: 121     Avg Loss: 0.253635
   Train Epoch: 141     Avg Loss: 0.227704
   Train Epoch: 161     Avg Loss: 0.243988
   Train Epoch: 181     Avg Loss: 0.271830
   Train Epoch: 201     Avg Loss: 0.247277
   Train Epoch: 221     Avg Loss: 0.245951
   Train Epoch: 241     Avg Loss: 0.169985
   Train Epoch: 261     Avg Loss: 0.221304
   Train Epoch: 281     Avg Loss: 0.193972
   Train Epoch: 301     Avg Loss: 0.163734
   Train Epoch: 321     Avg Loss: 0.131504
   Train Epoch: 341     Avg Loss: 0.126904
   Train Epoch: 361     Avg Loss: 0.121908
   Train Epoch: 381     Avg Loss: 0.121235
  Start fitting svm model with random search cv...
  SVC training done in 426.6363(s). Best Hyperparameters: {'kernel': 'poly', 'gamma': 'scale', 'C': 0.8291821660947628}
   Train Epoch: 400     Avg Loss: 0.120510
   Test set: Accuracy: 697/2000 (34.85%) - classifier: SVM - k-NN acc benchmark: 698/2000 (34.90%)
  Run execution time: train: 2806.435 (s) - eval: 3.760 (s)- total: 2810.195 (s)
  Run# [2/3] - Num Samples For Training 100 - Num Samples For Val 2000
   Train Epoch: 1       Avg Loss: 4.605818
   Train Epoch: 21      Avg Loss: 0.203616
   Train Epoch: 41      Avg Loss: 0.183776
   Train Epoch: 61      Avg Loss: 0.163109
   Train Epoch: 81      Avg Loss: 0.209487
   Train Epoch: 101     Avg Loss: 0.113210
   Train Epoch: 121     Avg Loss: 0.086554
   Train Epoch: 141     Avg Loss: 0.222571
```

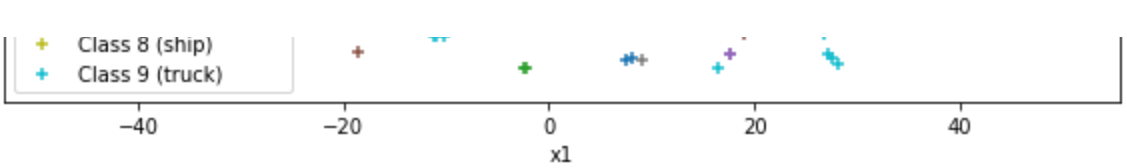
```
 Train Epoch: 161     Avg Loss: 0.245128
 Train Epoch: 181     Avg Loss: 0.117714
 Train Epoch: 201     Avg Loss: 0.207687
 Train Epoch: 221     Avg Loss: 0.101826
 Train Epoch: 241     Avg Loss: 0.202528
 Train Epoch: 261     Avg Loss: 0.121507
 Train Epoch: 281     Avg Loss: 0.588498
 Train Epoch: 301     Avg Loss: 0.073350
 Train Epoch: 321     Avg Loss: 0.055234
 Train Epoch: 341     Avg Loss: 0.056648
 Train Epoch: 361     Avg Loss: 0.051932
 Train Epoch: 381     Avg Loss: 0.043308
Start fitting svm model with random search cv...
SVC training done in 405.7248(s). Best Hyperparameters: {'kernel': 'poly', 'gamma': 'scale', 'C': 0.05873218708481509}
 Train Epoch: 400     Avg Loss: 0.045911
 Test set: Accuracy: 710/2000 (35.50%) - classifier: SVM - k-NN acc benchmark: 694/2000 (34.70%)
Run execution time: train: 2556.291 (s) - eval: 3.580 (s)- total: 2559.871 (s)
Run# [3/3] - Num Samples For Training 100 - Num Samples For Val 2000
 Train Epoch: 1       Avg Loss: 4.681615
 Train Epoch: 21      Avg Loss: 0.237048
 Train Epoch: 41      Avg Loss: 0.226676
 Train Epoch: 61      Avg Loss: 0.185654
 Train Epoch: 81      Avg Loss: 0.178795
 Train Epoch: 101     Avg Loss: 0.192540
 Train Epoch: 121     Avg Loss: 0.242457
 Train Epoch: 141     Avg Loss: 0.186452
 Train Epoch: 161     Avg Loss: 0.234093
 Train Epoch: 181     Avg Loss: 0.739190
 Train Epoch: 201     Avg Loss: 0.523965
 Train Epoch: 221     Avg Loss: 0.254948
 Train Epoch: 241     Avg Loss: 0.223038
 Train Epoch: 261     Avg Loss: 0.145437
 Train Epoch: 281     Avg Loss: 0.116713
 Train Epoch: 301     Avg Loss: 0.100633
 Train Epoch: 321     Avg Loss: 0.090826
 Train Epoch: 341     Avg Loss: 0.080493
 Train Epoch: 361     Avg Loss: 0.522418
 Train Epoch: 381     Avg Loss: 0.053824
Start fitting svm model with random search cv...
SVC training done in 410.8757(s). Best Hyperparameters: {'kernel': 'poly', 'gamma': 'scale', 'C': 74.42349368719803}
 Train Epoch: 400     Avg Loss: 0.056279
 Test set: Accuracy: 688/2000 (34.40%) - classifier: SVM - k-NN acc benchmark: 686/2000 (34.30%)
Run execution time: train: 2702.050 (s) - eval: 3.198 (s)- total: 2705.248 (s)
Scenario 1/1 - Epochs: 400 - lr: - 0.005 - dropout: 0 - Weight_decay: 1e-05 - Grad_clip: 0.005 - alpha: 0.3 - k=9
Avg execution time: train: 2688.259 +- 102.585 (s) - eval: 3.513 +- 0.235 (s) - total: 2691.771 +- 102.638 (s) on Tesla P100-PCIE-
16GB
Final acc over 3 instances: 34.92 +- 0.45
```



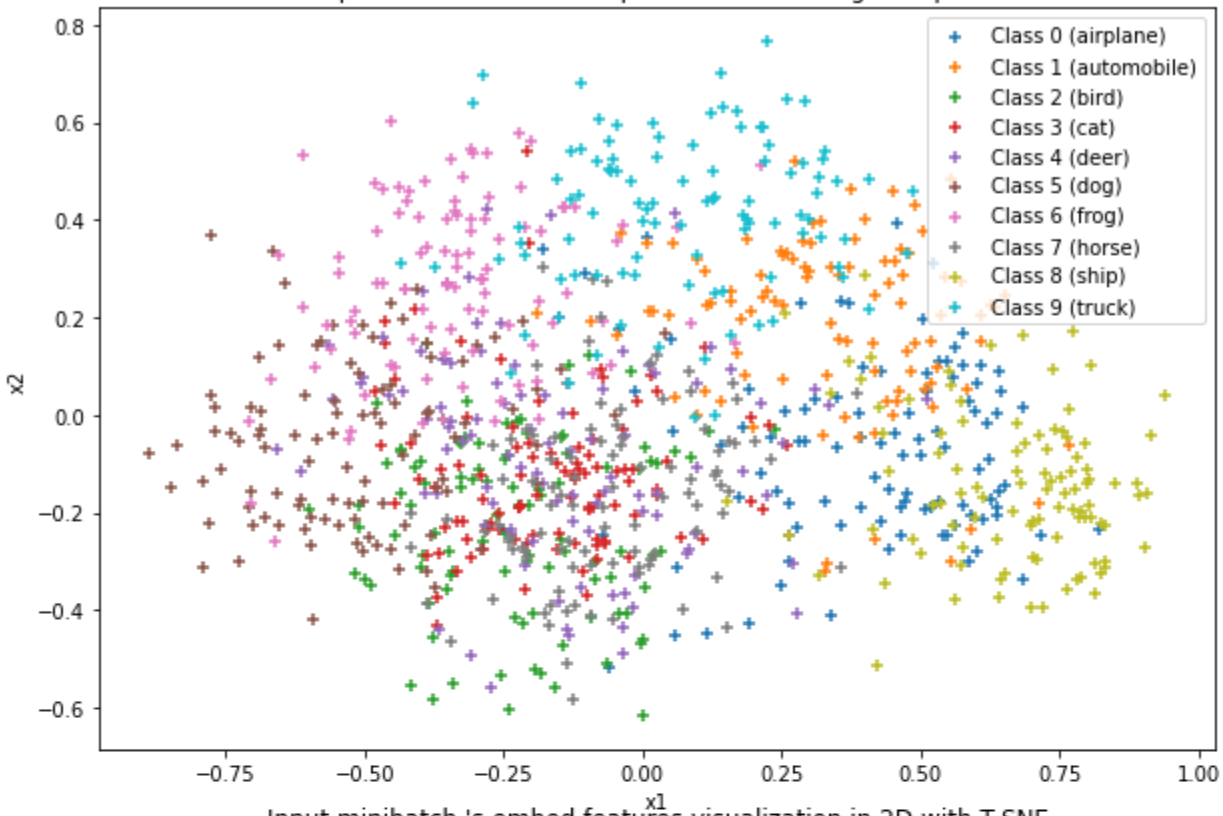


Input minibatch 's embed features visualization in 2D with T-SNE
Scenario 1/1 - Epochs: 400 - lr: - 0.005 - dropout: 0 - Weight_decay: 1e-05 - Grad_clip: 0.005 - alpha: 0.3 - k=9 - run: 1
Note: Epoch: 1 - Batch multiplier: x10 - hard negative prob: 0.10

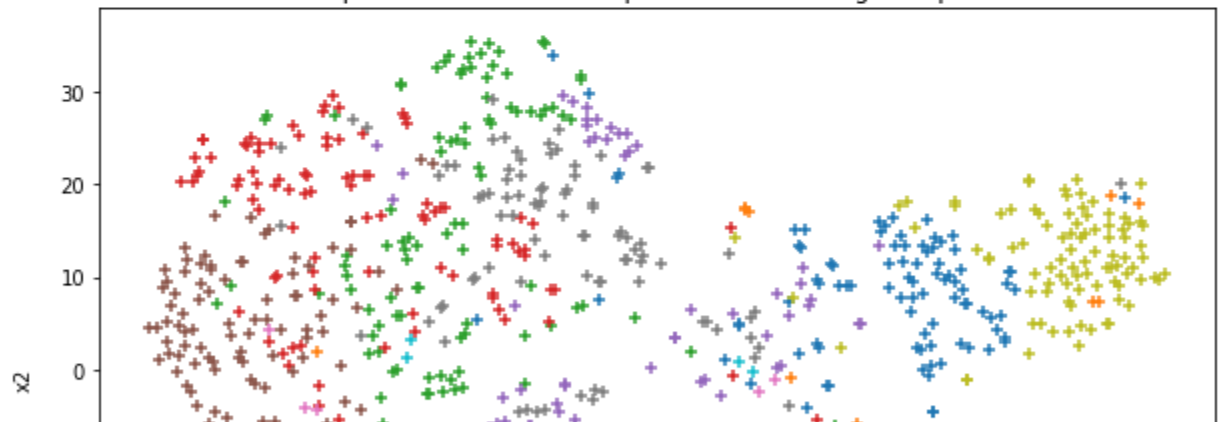


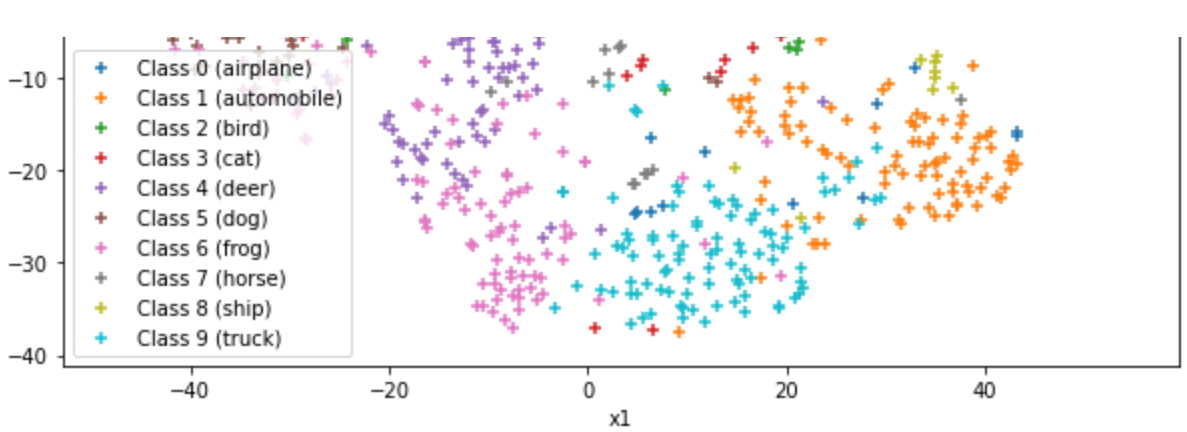


Input minibatch's embed features visualization in 2D with PCA transform
Scenario 1/1 - Epochs: 400 - lr: - 0.005 - dropout: 0 - Weight_decay: 1e-05 - Grad_clip: 0.005 - alpha: 0.3 - k=9 - run: 1
Note:Epoch: 400 - Batch multiplier: x10 - hard negative prob: 0.10

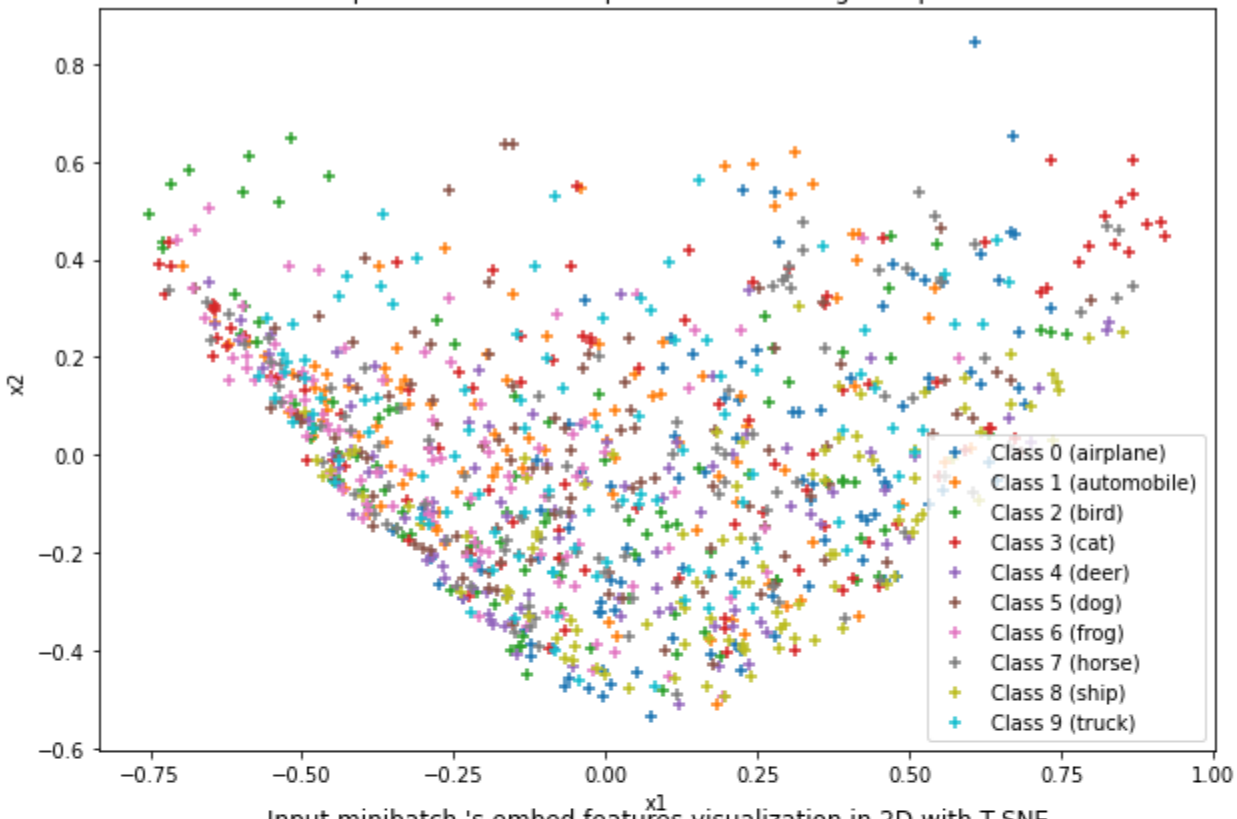


Input minibatch 's embed features visualization in 2D with T-SNE
Scenario 1/1 - Epochs: 400 - lr: - 0.005 - dropout: 0 - Weight_decay: 1e-05 - Grad_clip: 0.005 - alpha: 0.3 - k=9 - run: 1
Note: Epoch: 400 - Batch multiplier: x10 - hard negative prob: 0.10

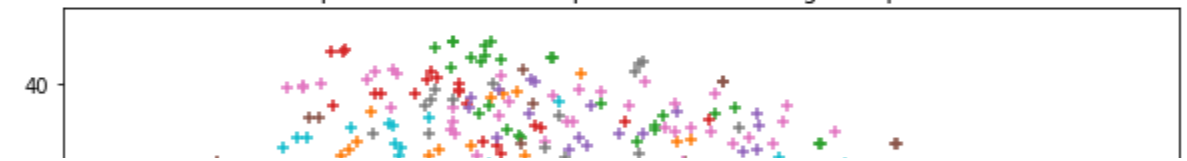


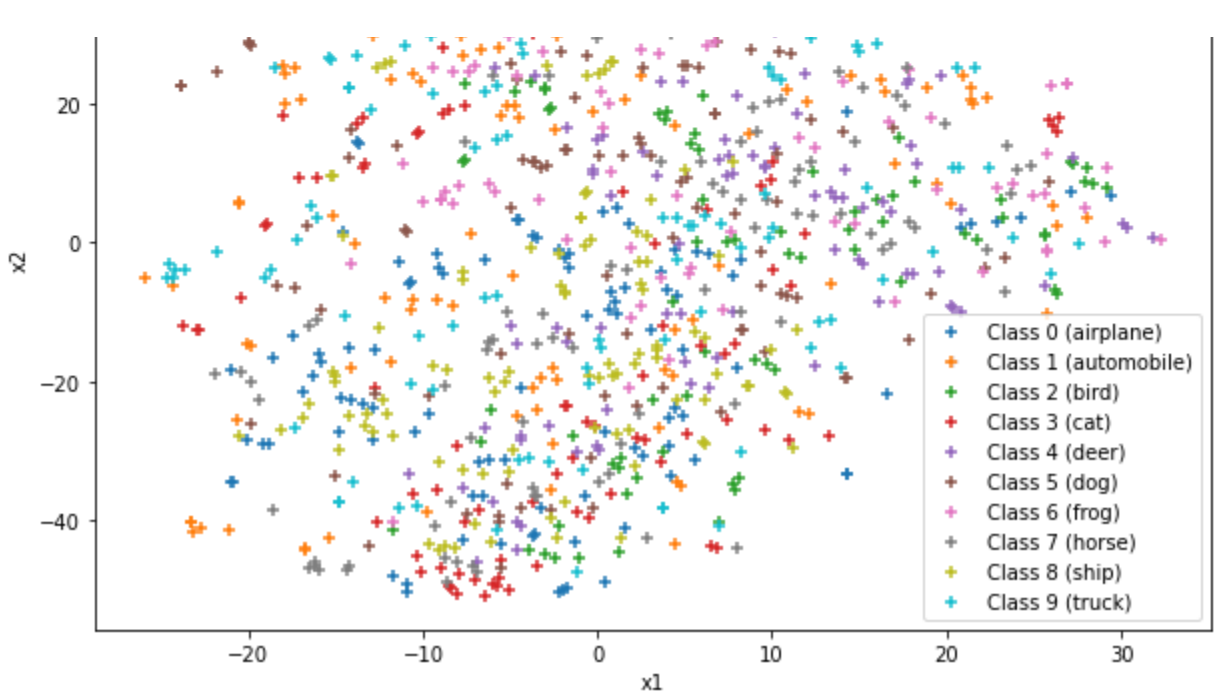


Input minibatch's embed features visualization in 2D with PCA transform
Scenario 1/1 - Epochs: 400 - lr: - 0.005 - dropout: 0 - Weight_decay: 1e-05 - Grad_clip: 0.005 - alpha: 0.3 - k=9 - run: 2
Note:Epoch: 1 - Batch multiplier: x10 - hard negative prob: 0.10

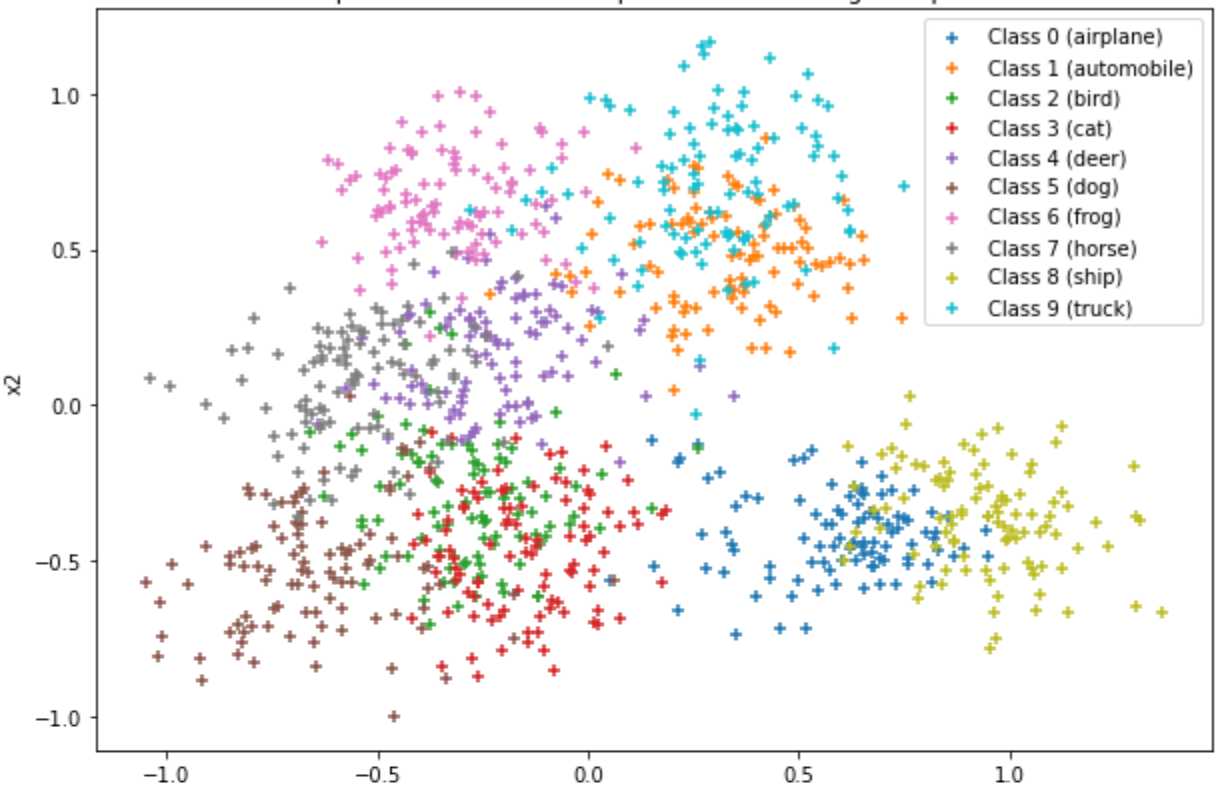


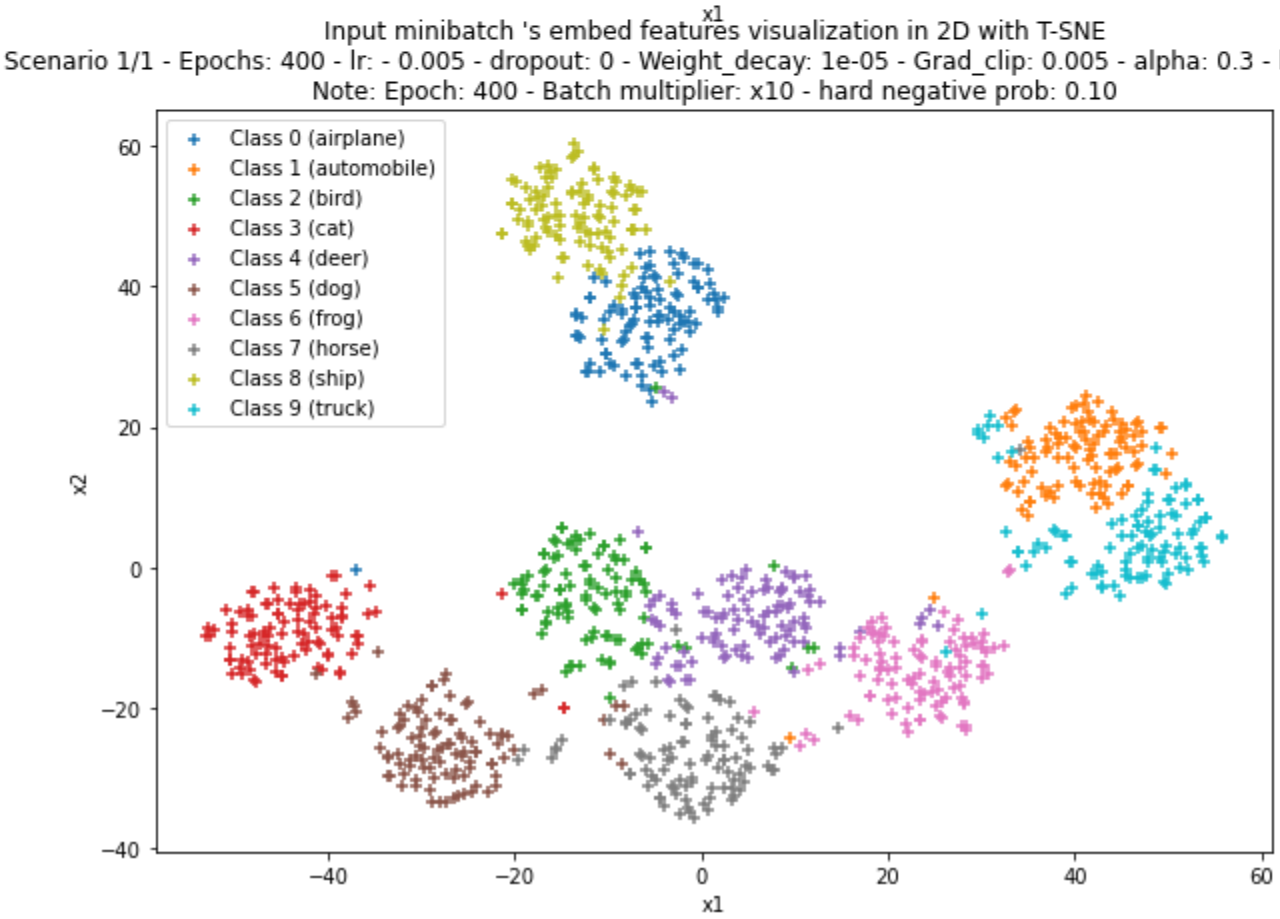
Input minibatch 's embed features visualization in 2D with T-SNE
Scenario 1/1 - Epochs: 400 - lr: - 0.005 - dropout: 0 - Weight_decay: 1e-05 - Grad_clip: 0.005 - alpha: 0.3 - k=9 - run: 2
Note: Epoch: 1 - Batch multiplier: x10 - hard negative prob: 0.10

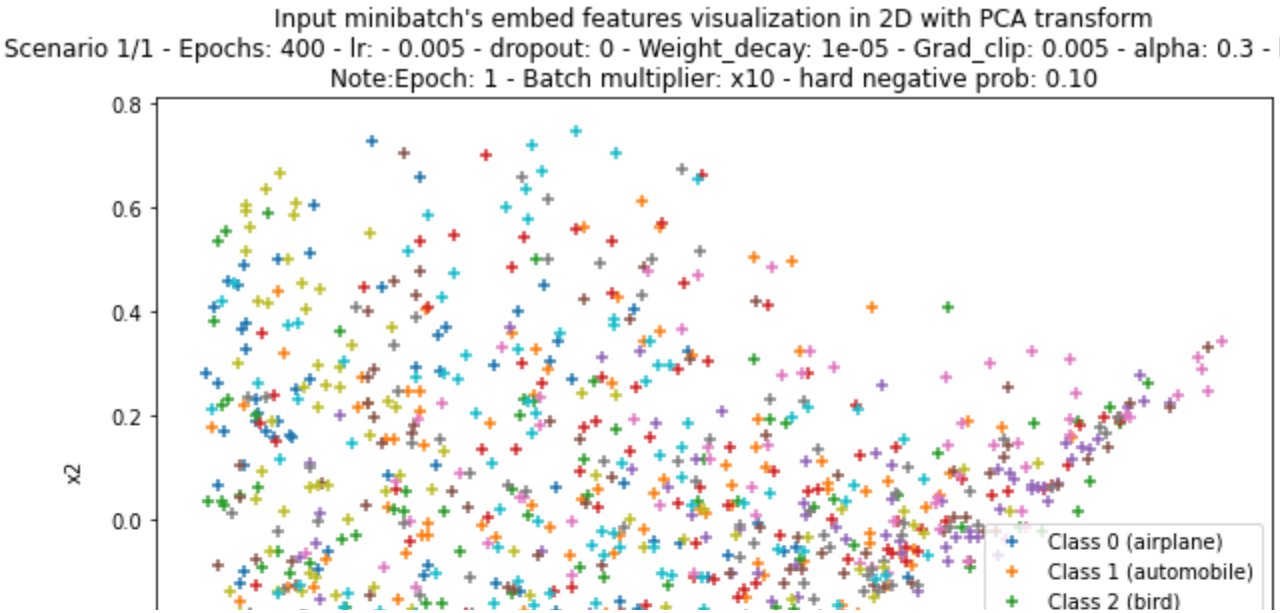


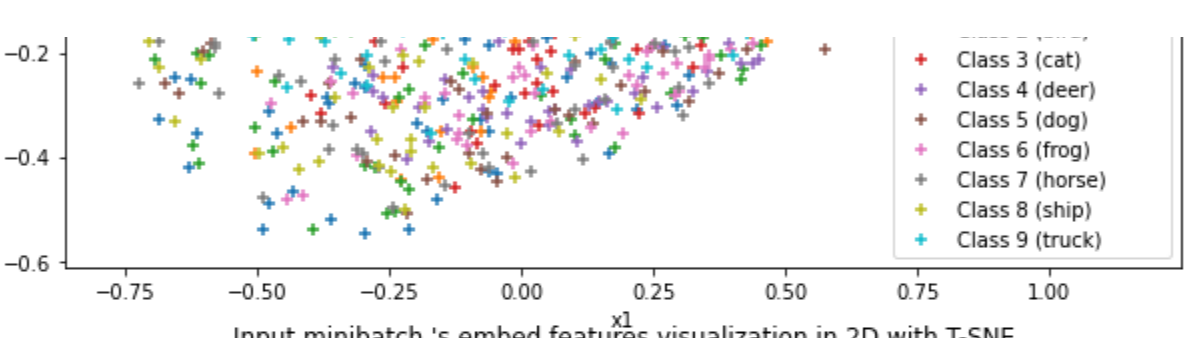


Input minibatch's embed features visualization in 2D with PCA transform
Scenario 1/1 - Epochs: 400 - lr: - 0.005 - dropout: 0 - Weight_decay: 1e-05 - Grad_clip: 0.005 - alpha: 0.3 - k=9 - run: 2
Note:Epoch: 400 - Batch multiplier: x10 - hard negative prob: 0.10

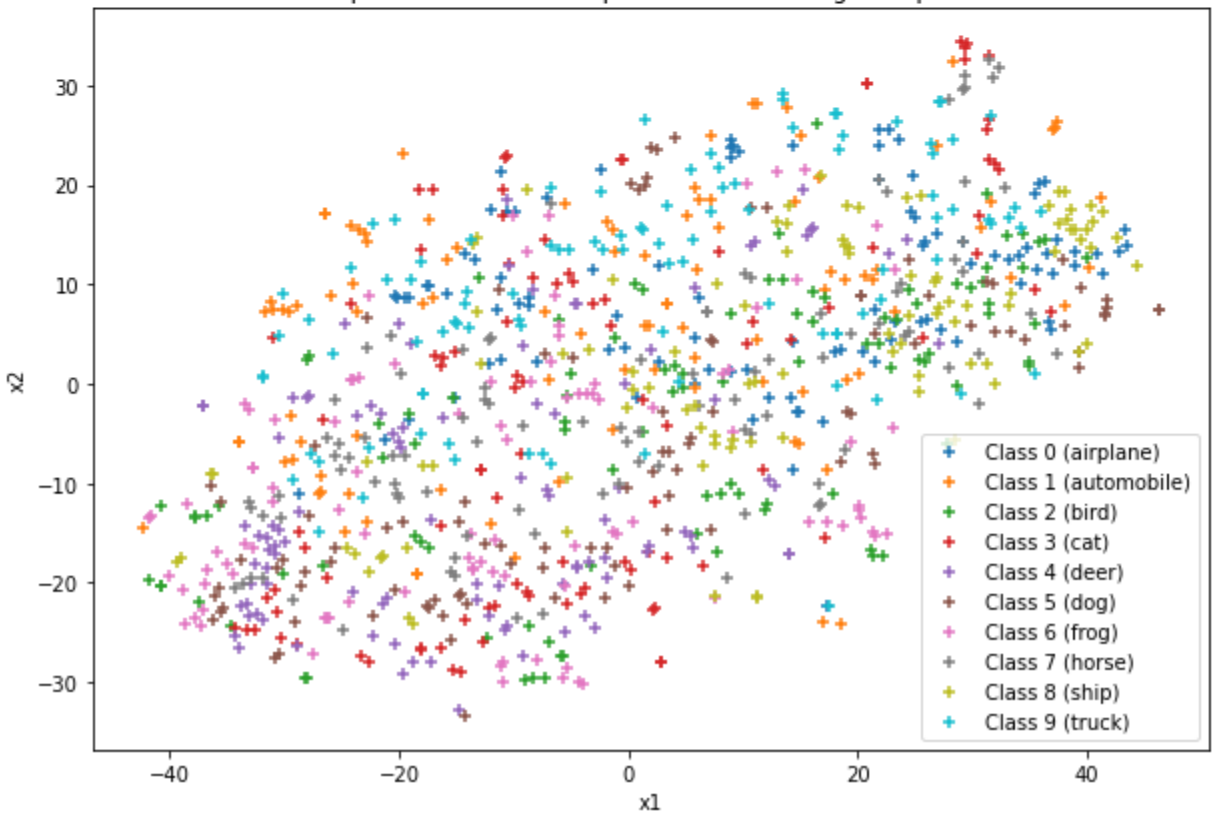



x1
Input minibatch 's embed features visualization in 2D with T-SNE
Scenario 1/1 - Epochs: 400 - lr: - 0.005 - dropout: 0 - Weight_decay: 1e-05 - Grad_clip: 0.005 - alpha: 0.3 - k=9 - run: 2
Note: Epoch: 400 - Batch multiplier: x10 - hard negative prob: 0.10
Class 0 (airplane)
Class 1 (automobile)
Class 2 (bird)
Class 3 (cat)
Class 4 (deer)
Class 5 (dog)
Class 6 (frog)
Class 7 (horse)
Class 8 (ship)
Class 9 (truck)
x2
x1

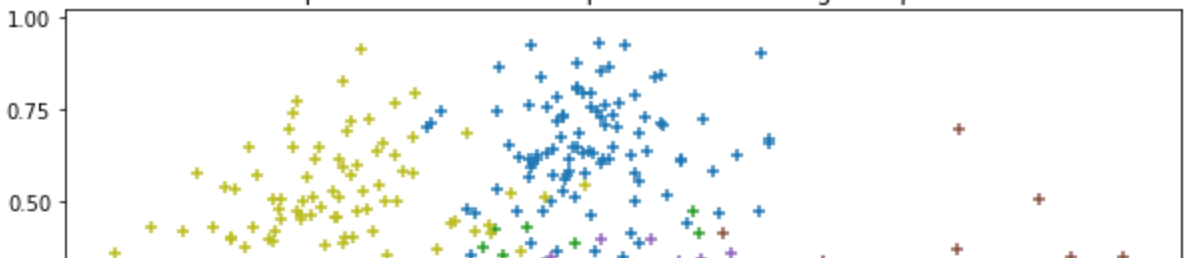

Input minibatch's embed features visualization in 2D with PCA transform
Scenario 1/1 - Epochs: 400 - lr: - 0.005 - dropout: 0 - Weight_decay: 1e-05 - Grad_clip: 0.005 - alpha: 0.3 - k=9 - run: 3
Note:Epoch: 1 - Batch multiplier: x10 - hard negative prob: 0.10
Class 0 (airplane)
Class 1 (automobile)
Class 2 (bird)
x2

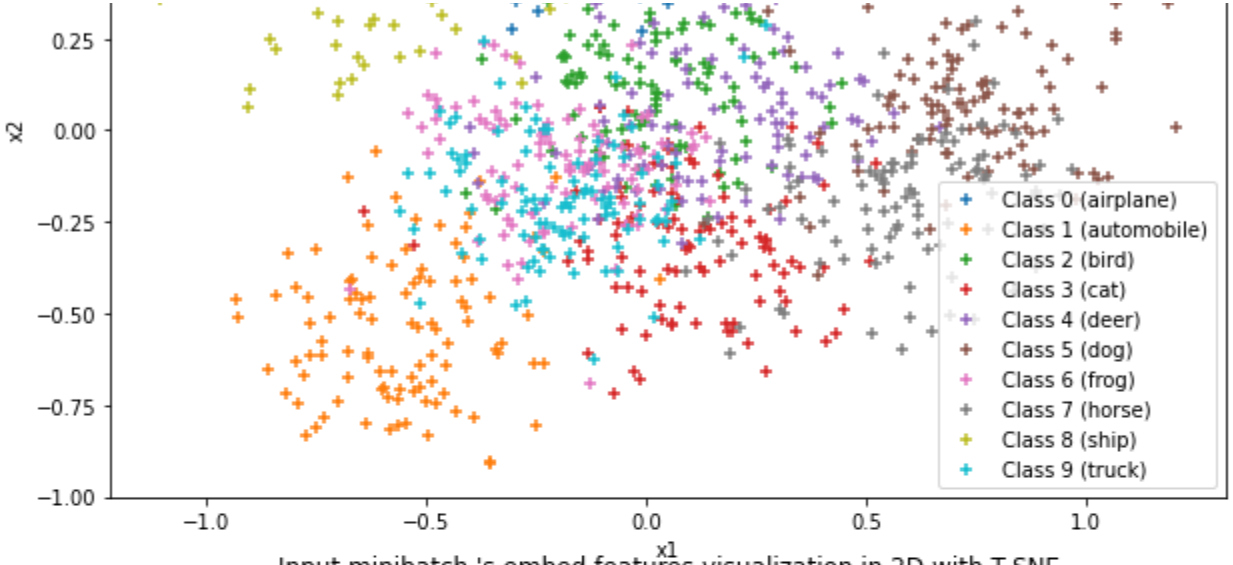


Input minibatch 's embed features visualization in 2D with T-SNE
Scenario 1/1 - Epochs: 400 - lr: - 0.005 - dropout: 0 - Weight_decay: 1e-05 - Grad_clip: 0.005 - alpha: 0.3 - k=9 - run: 3
Note: Epoch: 1 - Batch multiplier: x10 - hard negative prob: 0.10

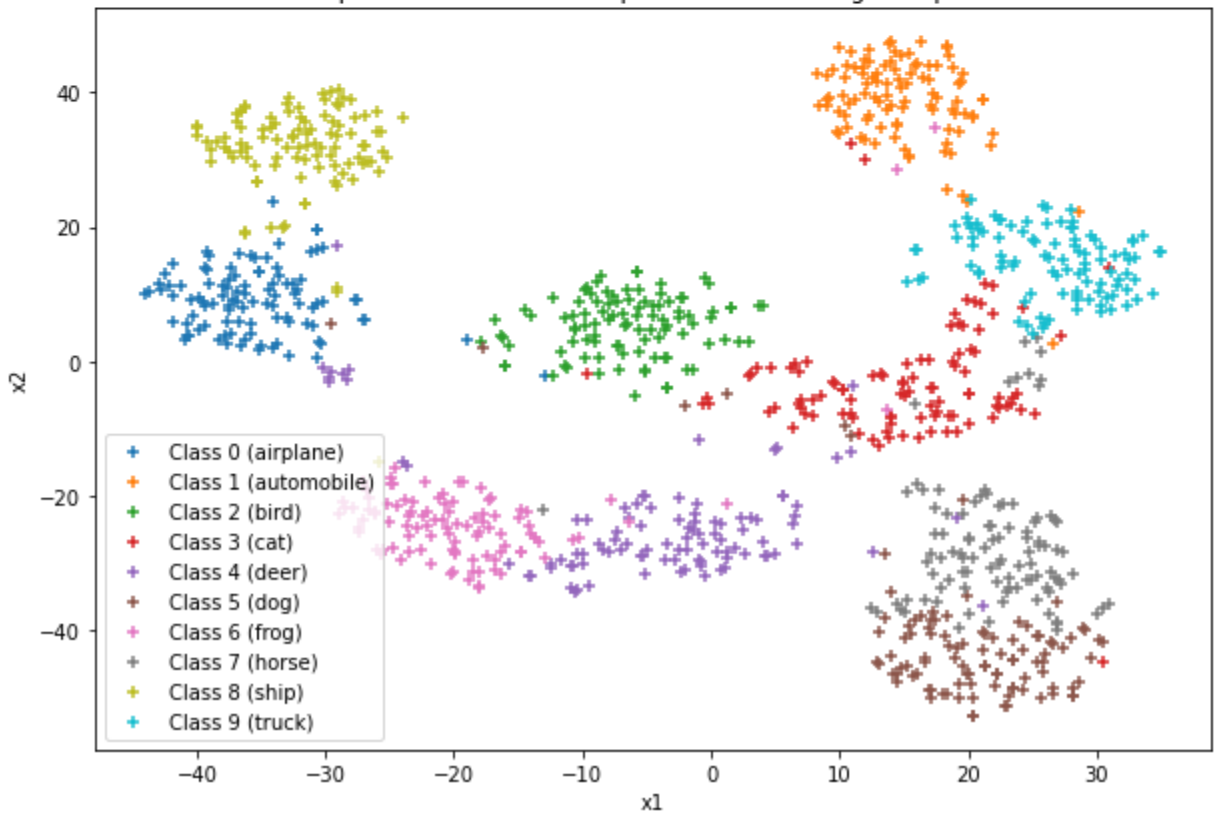


Input minibatch's embed features visualization in 2D with PCA transform
Scenario 1/1 - Epochs: 400 - lr: - 0.005 - dropout: 0 - Weight_decay: 1e-05 - Grad_clip: 0.005 - alpha: 0.3 - k=9 - run: 3
Note:Epoch: 400 - Batch multiplier: x10 - hard negative prob: 0.10





Input minibatch 's embed features visualization in 2D with T-SNE
Scenario 1/1 - Epochs: 400 - lr: - 0.005 - dropout: 0 - Weight_decay: 1e-05 - Grad_clip: 0.005 - alpha: 0.3 - k=9 - run: 3
Note: Epoch: 400 - Batch multiplier: x10 - hard negative prob: 0.10



Final_EfficientNetB0

May 2, 2021

[1]:
```
import os.path
from os import path
if path.exists("/content/cifar10_pytorch") == False:
  !git clone https://github.com/hussien/cifar10_pytorch.git
```

```
Cloning into 'cifar10_pytorch'...
remote: Enumerating objects: 96, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (9/9), done.
remote: Total 96 (delta 0), reused 3 (delta 0), pack-reused 87
Unpacking objects: 100% (96/96), done.
Checking out files: 100% (44/44), done.
```

[2]:
```
!pip install wget
```

```
Collecting wget
  Downloading https://files.pythonhosted.org/packages/47/6a/62e288da7bcda82b935f
f0c6cfe542970f04e29c756b0e147251b2fb251f/wget-3.2.zip
Building wheels for collected packages: wget
  Building wheel for wget (setup.py) ... done
  Created wheel for wget: filename=wget-3.2-cp37-none-any.whl size=9681
sha256=411dad8009f7d9634b4618513a9c3ff455052fb364162b183f6f06be03aa7552
  Stored in directory: /root/.cache/pip/wheels/40/15/30/7d8f7cea2902b4db79e3fea5
50d7d7b85ecb27ef992b618f3f
Successfully built wget
Installing collected packages: wget
Successfully installed wget-3.2
```

[ ]:
```
# if path.exists("/content/CINIC-10.zip") == False:
#   !wget -O 'CINIC-10.zip' 'https://storage.googleapis.com/kaggle-data-sets/
 ↪362150/707954/bundle/archive.zip?
 ↪X-Goog-Algorithm=GOOG4-RSA-SHA256&X-Goog-Credential=gcp-kaggle-com%40kaggle-161607.
 ↪iam.gserviceaccount.
 ↪com%2F20210430%2Fauto%2Fstorage%2Fgoog4_request&X-Goog-Date=20210430T012007Z&X-Goog-Expires
```

[ ]:
```
# if path.exists("/content/CINIC-10") == False:
#   !unzip -q  '/content/CINIC-10.zip' -d '/content/CINIC-10'
```

[ ]:
```
# from google.colab import drive
# drive.mount('/content/gdrive')
```

[3]:
```
import numpy as np
%matplotlib inline
import matplotlib.pyplot as plt
from PIL import Image
import torch
from torchvision import datasets, models, transforms
import torch.nn as nn
from torch.nn import functional as F
import torch.optim as optim

import os
import torch
import torchvision
import tarfile
import numpy as np
import torch.nn.functional as F
from torchvision.datasets.utils import download_url
from torchvision.datasets import ImageFolder
from torch.utils.data import DataLoader
import torchvision.transforms as tt
from torch.utils.data import random_split
from torchvision.utils import make_grid
import matplotlib
import matplotlib.pyplot as plt
```

[4]:
```
!pip install yacs
```

```
Collecting yacs
  Downloading https://files.pythonhosted.org/packages/38/4f/fe9a4d472aa867878ce3
bb7efb16654c5d63672b86dc0e6e953a67018433/yacs-0.1.8-py3-none-any.whl
Requirement already satisfied: PyYAML in /usr/local/lib/python3.7/dist-packages
(from yacs) (3.13)
Installing collected packages: yacs
Successfully installed yacs-0.1.8
```

[5]:
```
import os
import argparse
import torch
import os, sys
from cifar10_pytorch.cifar10_pytorch.engine import train, test
from cifar10_pytorch.cifar10_pytorch.config import cfg
from cifar10_pytorch.cifar10_pytorch.modeling import build_model
```

[ ]:
```
# ! rm -r 'CINIC-10-Filtered_1K'
```

[6]:
```
if path.exists("/content/CINIC-10-Filtered_1K") == False:
  !unzip -q  '/content/cifar10_pytorch/CINIC-10-Filtered_1K.zip' -d '/content/'
if path.exists("train") == False:
  !unzip -q  '/content/cifar10_pytorch/train.zip' -d '/content/'
if path.exists("/content/test") == False:
  !unzip -q  '/content/cifar10_pytorch/test.zip' -d '/content/'
```

[7]:
```
def arg_parser():
    parser = argparse.ArgumentParser(description="CIFAR10 training")
    parser.add_argument(
        "--config",
        default='/content/cifar10_pytorch/configs/efficientnetB0.yaml',
        # default='/content/cifar10_pytorch/configs/efficientnetB4.yaml',
        help="path to config file",
        type=str
    )
    parser.add_argument(
        '--tfboard', help='tensorboard path for logging', type=str,␣
 ↪default='out')
    parser.add_argument('--checkpoint_dir', type=str,
                        default='checkpoints',
                        help='directory where checkpoint files are saved')
    parser.add_argument('--resume', type=str,
                        default=None,
                        help='checkpoint file path')
    return parser.parse_args("")
```

[8]:
```
!pip install tensorboardX
```

```
Collecting tensorboardX
  Downloading https://files.pythonhosted.org/packages/07/84/46421bd3e0e89a
92682b1a38b40efc22dafb6d8e3d947e4ceefd4a5fabc7/tensorboardX-2.2-py2.py3-none-
any.whl (120kB)
     || 122kB 7.9MB/s
Requirement already satisfied: protobuf>=3.8.0 in /usr/local/lib/python3.7
/dist-packages (from tensorboardX) (3.12.4)
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages
(from tensorboardX) (1.19.5)
Requirement already satisfied: setuptools in /usr/local/lib/python3.7/dist-
packages (from protobuf>=3.8.0->tensorboardX) (56.0.0)
Requirement already satisfied: six>=1.9 in /usr/local/lib/python3.7/dist-
packages (from protobuf>=3.8.0->tensorboardX) (1.15.0)
Installing collected packages: tensorboardX
Successfully installed tensorboardX-2.2
```

[9]:
```
# args and configs
args = arg_parser()
if args.config:
```

```
    cfg.merge_from_file(args.config)
cfg.freeze()
os.makedirs(args.checkpoint_dir, exist_ok=True)

# Model definition
# cfg.SOLVER.BATCHSIZE=64
model = build_model(cfg)
# cfg.MODEL.DEVICE='cpu'
print(cfg.MODEL.DEVICE)
device = cfg.MODEL.DEVICE
model.to(device)

# Optimizer settings
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(),
                            lr=cfg.SOLVER.BASE_LR,
                            momentum=cfg.SOLVER.MOMENTUM,
                            weight_decay=cfg.SOLVER.WEIGHT_DECAY,
                            )

scheduler = torch.optim.lr_scheduler.MultiStepLR(
    optimizer,
    cfg.SOLVER.MILESTONES,
    cfg.SOLVER.GAMMA,
)
```

```
Downloading: "https://github.com/lukemelas/EfficientNet-
PyTorch/releases/download/1.0/efficientnet-b0-355c32eb.pth" to
/root/.cache/torch/hub/checkpoints/efficientnet-b0-355c32eb.pth
```

```
HBox(children=(FloatProgress(value=0.0, max=21388428.0), HTML(value='')))
```

```
cuda
```

[10]:
```
import torch
import torchvision
from torchvision import datasets,transforms
import torchvision.transforms as T

def build_transforms(cfg, is_train=True):
    list_transforms = list()

    # Resizing. Some models must have ImageNet-size images as input
    if is_train and cfg.DATA.IMG_SIZE != 32:
        list_transforms.append(T.Resize((cfg.DATA.IMG_SIZE, cfg.DATA.IMG_SIZE)))
    elif is_train == False and cfg.TEST.IMG_SIZE != 32:
```

```
        list_transforms.append(T.Resize((cfg.DATA.IMG_SIZE, cfg.DATA.IMG_SIZE)))

    if is_train:
        if cfg.DATA.RANDOMCROP:
            list_transforms.append(T.RandomCrop(cfg.DATA.IMG_SIZE, padding=4))
        if cfg.DATA.LRFLIP:
            list_transforms.append(T.RandomHorizontalFlip())

    list_transforms.extend([
        T.ToTensor(),
        T.Normalize(cfg.DATA.NORMALIZE_MEAN, cfg.DATA.NORMALIZE_STD),
    ])

    transforms = T.Compose(list_transforms)

    return transforms
```

[11]:
```
from numpy.random import RandomState
from torch.utils.data import Subset

def prepare_cifar10_dataset(cfg):
    """ prepare CIFAR10 dataset based on configuration"""

    transform_train = build_transforms(cfg, is_train=True)
    transform_test = build_transforms(cfg, is_train=False)


    normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                     std=[0.229, 0.224, 0.225])

    # dataset_train = torchvision.datasets.CIFAR10(
    #     root='./data',
    #     train=True,
    #     download=True,
    #     transform=transform_train
    # )
    dataset_train = datasets.ImageFolder(
            '/content/CINIC-10-Filtered_1K/train',
            transform=transform_train
    )
    dataloader_train = torch.utils.data.DataLoader(
        dataset_train,
        batch_size=cfg.SOLVER.BATCHSIZE,
        shuffle=True,
        num_workers=cfg.SOLVER.NUM_WORKERS
    )
    dataset_valid = torchvision.datasets.CIFAR10(
```

```
        root='./data',
        train=False,
        download=True,
        transform=transform_test
    )

    prng = RandomState(10)
    random_permute = prng.permutation(np.arange(0, 1000))
    print(len(dataset_valid.targets))
    indx_val = np.concatenate([np.where(np.array(dataset_valid.targets) ==␣
↪classe)[0][random_permute[10:210]] for classe in range(0, 10)])
    val_data = Subset(dataset_valid, indx_val)

    print('Num Samples For Val= %d'%(val_data.indices.shape[0]))

    dataloader_valid = torch.utils.data.DataLoader(
        val_data,
        batch_size=cfg.TEST.BATCHSIZE,
        shuffle=False,
        num_workers=cfg.SOLVER.NUM_WORKERS
    )

    return dataloader_train, dataloader_valid
```

[12]:
```
dataloader_train, dataloader_test = prepare_cifar10_dataset(cfg)
# print(len(dataloader_train.dataset))
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to
./data/cifar-10-python.tar.gz

HBox(children=(FloatProgress(value=0.0, max=170498071.0), HTML(value='')))



Extracting ./data/cifar-10-python.tar.gz to ./data
10000
Num Samples For Val= 2000
```

training settings:
Model: PRETRAINED: "EfficientNetB0" DROPOUT_RATE: 0.2
Augmentation: IMG_SIZE: 224 RANDOMCROP: True
SOLVER: LR: 0.015 BATCHSIZE: 64 END_EPOCH : 30 WEIGHT_DECAY: 0.0001

[13]:
```
# if args.resume:
#     checkpoint = torch.load(args.resume)
#     model.load_state_dict(checkpoint['model_state_dict'])
#     optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
#     scheduler.load_state_dict(checkpoint['scheduler_state_dict'])
#     #scheduler.last_epoch = checkpoint['epoch']
```

```
#     start_epoch = checkpoint['epoch']
# else:
#     start_epoch = 0

end_epoch = 30
train_loss_epoch,valid_loss_epoch=[],[]
train_acc_epoch,valid_acc_epoch=[],[]
for epoch in range(0, end_epoch):
    scheduler.step()
    print("------ start epoch {} / {} with lr = {:.5f} -----".format(
        epoch+1, end_epoch, scheduler.get_lr()[0])
    )
    train_loss, train_acc = train(model, device, dataloader_train, criterion,␣
 ↪optimizer, epoch)
    train_loss_epoch.append(train_loss)
    train_acc_epoch.append(train_acc)
    # if epoch%5 == 0:
    valid_loss, valid_acc = test(model, device, dataloader_test, criterion,␣
 ↪epoch)
    valid_loss_epoch.append(valid_loss)
    valid_acc_epoch.append(valid_acc)
```

```
/usr/local/lib/python3.7/dist-packages/torch/optim/lr_scheduler.py:134:
UserWarning: Detected call of `lr_scheduler.step()` before `optimizer.step()`.
In PyTorch 1.1.0 and later, you should call them in the opposite order:
`optimizer.step()` before `lr_scheduler.step()`.  Failure to do this will result
in PyTorch skipping the first value of the learning rate schedule. See more
details at https://pytorch.org/docs/stable/optim.html#how-to-adjust-learning-
rate
  "https://pytorch.org/docs/stable/optim.html#how-to-adjust-learning-rate",
UserWarning)
/usr/local/lib/python3.7/dist-packages/torch/optim/lr_scheduler.py:417:
UserWarning: To get the last learning rate computed by the scheduler, please use
`get_last_lr()`.
  "please use `get_last_lr()`.", UserWarning)

------ start epoch 1 / 30 with lr = 0.01500 -----

train set size= 158  batch size= 64
train-> epoch 1, iter 100 / 158, Loss: 1.169 | Acc: 62.453

test set size= 32  batch size= 64
Accuracy: 77.4500
------ start epoch 2 / 30 with lr = 0.01500 -----

train set size= 158  batch size= 64
train-> epoch 2, iter 100 / 158, Loss: 0.514 | Acc: 81.734
```

```
test set size= 32  batch size= 64
Accuracy: 78.7500
------ start epoch 3 / 30 with lr = 0.01500 -----

train set size= 158  batch size= 64
train-> epoch 3, iter 100 / 158, Loss: 0.365 | Acc: 86.984

test set size= 32  batch size= 64
Accuracy: 80.8000
------ start epoch 4 / 30 with lr = 0.01500 -----

train set size= 158  batch size= 64
train-> epoch 4, iter 100 / 158, Loss: 0.276 | Acc: 90.594

test set size= 32  batch size= 64
Accuracy: 76.9000
------ start epoch 5 / 30 with lr = 0.01500 -----

train set size= 158  batch size= 64
train-> epoch 5, iter 100 / 158, Loss: 0.200 | Acc: 93.203

test set size= 32  batch size= 64
Accuracy: 82.3000
------ start epoch 6 / 30 with lr = 0.01500 -----

train set size= 158  batch size= 64
train-> epoch 6, iter 100 / 158, Loss: 0.157 | Acc: 94.469

test set size= 32  batch size= 64
Accuracy: 78.4000
------ start epoch 7 / 30 with lr = 0.01500 -----

train set size= 158  batch size= 64
train-> epoch 7, iter 100 / 158, Loss: 0.134 | Acc: 95.359

test set size= 32  batch size= 64
Accuracy: 78.9000
------ start epoch 8 / 30 with lr = 0.01500 -----

train set size= 158  batch size= 64
train-> epoch 8, iter 100 / 158, Loss: 0.111 | Acc: 96.281

test set size= 32  batch size= 64
Accuracy: 82.9000
------ start epoch 9 / 30 with lr = 0.01500 -----

train set size= 158  batch size= 64
```

```
train-> epoch 9, iter 100 / 158, Loss: 0.102 | Acc: 96.703

test set size= 32  batch size= 64
Accuracy: 82.8000
------ start epoch 10 / 30 with lr = 0.01500 -----

train set size= 158  batch size= 64
train-> epoch 10, iter 100 / 158, Loss: 0.083 | Acc: 97.438

test set size= 32  batch size= 64
Accuracy: 85.2500
------ start epoch 11 / 30 with lr = 0.01500 -----

train set size= 158  batch size= 64
train-> epoch 11, iter 100 / 158, Loss: 0.077 | Acc: 97.562

test set size= 32  batch size= 64
Accuracy: 84.4000
------ start epoch 12 / 30 with lr = 0.01500 -----

train set size= 158  batch size= 64
train-> epoch 12, iter 100 / 158, Loss: 0.054 | Acc: 98.156

test set size= 32  batch size= 64
Accuracy: 81.8000
------ start epoch 13 / 30 with lr = 0.01500 -----

train set size= 158  batch size= 64
train-> epoch 13, iter 100 / 158, Loss: 0.059 | Acc: 97.969

test set size= 32  batch size= 64
Accuracy: 83.5500
------ start epoch 14 / 30 with lr = 0.01500 -----

train set size= 158  batch size= 64
train-> epoch 14, iter 100 / 158, Loss: 0.062 | Acc: 97.844

test set size= 32  batch size= 64
Accuracy: 84.4500
------ start epoch 15 / 30 with lr = 0.01500 -----

train set size= 158  batch size= 64
train-> epoch 15, iter 100 / 158, Loss: 0.052 | Acc: 98.281

test set size= 32  batch size= 64
Accuracy: 83.8000
------ start epoch 16 / 30 with lr = 0.01500 -----
```

```
train set size= 158  batch size= 64
train-> epoch 16, iter 100 / 158, Loss: 0.046 | Acc: 98.484

test set size= 32  batch size= 64
Accuracy: 84.1000
------ start epoch 17 / 30 with lr = 0.01500 -----

train set size= 158  batch size= 64
train-> epoch 17, iter 100 / 158, Loss: 0.039 | Acc: 98.672

test set size= 32  batch size= 64
Accuracy: 85.3000
------ start epoch 18 / 30 with lr = 0.01500 -----

train set size= 158  batch size= 64
train-> epoch 18, iter 100 / 158, Loss: 0.042 | Acc: 98.547

test set size= 32  batch size= 64
Accuracy: 84.8500
------ start epoch 19 / 30 with lr = 0.01500 -----

train set size= 158  batch size= 64
train-> epoch 19, iter 100 / 158, Loss: 0.036 | Acc: 98.719

test set size= 32  batch size= 64
Accuracy: 86.1500
------ start epoch 20 / 30 with lr = 0.00015 -----

train set size= 158  batch size= 64
train-> epoch 20, iter 100 / 158, Loss: 0.032 | Acc: 98.922

test set size= 32  batch size= 64
Accuracy: 86.9000
------ start epoch 21 / 30 with lr = 0.00150 -----

train set size= 158  batch size= 64
train-> epoch 21, iter 100 / 158, Loss: 0.024 | Acc: 99.250

test set size= 32  batch size= 64
Accuracy: 87.1000
------ start epoch 22 / 30 with lr = 0.00150 -----

train set size= 158  batch size= 64
train-> epoch 22, iter 100 / 158, Loss: 0.022 | Acc: 99.328

test set size= 32  batch size= 64
Accuracy: 87.3500
------ start epoch 23 / 30 with lr = 0.00150 -----
```

```
train set size= 158  batch size= 64
train-> epoch 23, iter 100 / 158, Loss: 0.019 | Acc: 99.438

test set size= 32  batch size= 64
Accuracy: 87.3500
------ start epoch 24 / 30 with lr = 0.00150 -----

train set size= 158  batch size= 64
train-> epoch 24, iter 100 / 158, Loss: 0.018 | Acc: 99.359

test set size= 32  batch size= 64
Accuracy: 87.4000
------ start epoch 25 / 30 with lr = 0.00150 -----

train set size= 158  batch size= 64
train-> epoch 25, iter 100 / 158, Loss: 0.015 | Acc: 99.500

test set size= 32  batch size= 64
Accuracy: 87.6000
------ start epoch 26 / 30 with lr = 0.00150 -----

train set size= 158  batch size= 64
train-> epoch 26, iter 100 / 158, Loss: 0.020 | Acc: 99.406

test set size= 32  batch size= 64
Accuracy: 87.5000
------ start epoch 27 / 30 with lr = 0.00150 -----

train set size= 158  batch size= 64
train-> epoch 27, iter 100 / 158, Loss: 0.017 | Acc: 99.609

test set size= 32  batch size= 64
Accuracy: 87.4000
------ start epoch 28 / 30 with lr = 0.00150 -----

train set size= 158  batch size= 64
train-> epoch 28, iter 100 / 158, Loss: 0.015 | Acc: 99.562

test set size= 32  batch size= 64
Accuracy: 87.0500
------ start epoch 29 / 30 with lr = 0.00150 -----

train set size= 158  batch size= 64
train-> epoch 29, iter 100 / 158, Loss: 0.016 | Acc: 99.625

test set size= 32  batch size= 64
Accuracy: 87.0500
```

```
------ start epoch 30 / 30 with lr = 0.00002 -----

train set size= 158  batch size= 64
train-> epoch 30, iter 100 / 158, Loss: 0.016 | Acc: 99.641

test set size= 32  batch size= 64
Accuracy: 87.2500
```

Save Traind Model

[14]:
```
import datetime
currentDT = datetime.datetime.now()
torch.save({'epoch': epoch + 1,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'scheduler_state_dict': scheduler.state_dict(),
            'train_loss_epoch': train_loss_epoch,
            'test_loss_epoch': valid_loss_epoch,
            'train_acc_epoch': train_acc_epoch,
            'test_acc_epoch': valid_acc_epoch
            },
            os.path.join("/content/",␣
 ↪"EfficientNetB0_CINIC10_30epoch_"+str(currentDT)+".ckpt"))
```

Load Trained Model

[ ]:
```
# checkpoint = torch.load('/content/cifar10_pytorch/ModelPickles/
 ↪EfficientNetB0_CINIC10_30epoch_20210430_R3.ckpt')
# model.load_state_dict(checkpoint['model_state_dict'])
# optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
# epoch = checkpoint['epoch']
# train_loss_epoch = checkpoint['train_loss_epoch']
# valid_loss_epoch=checkpoint['test_loss_epoch']
# train_acc_epoch=checkpoint['train_acc_epoch']
# valid_acc_epoch=checkpoint['test_acc_epoch']
```

[15]:
```
accs = np.array(valid_acc_epoch)
print('Acc over 2 instances: %.2f +- %.2f'%(accs.mean(),accs.std()))
```

```
Acc over 2 instances: 84.09 +- 3.30
```

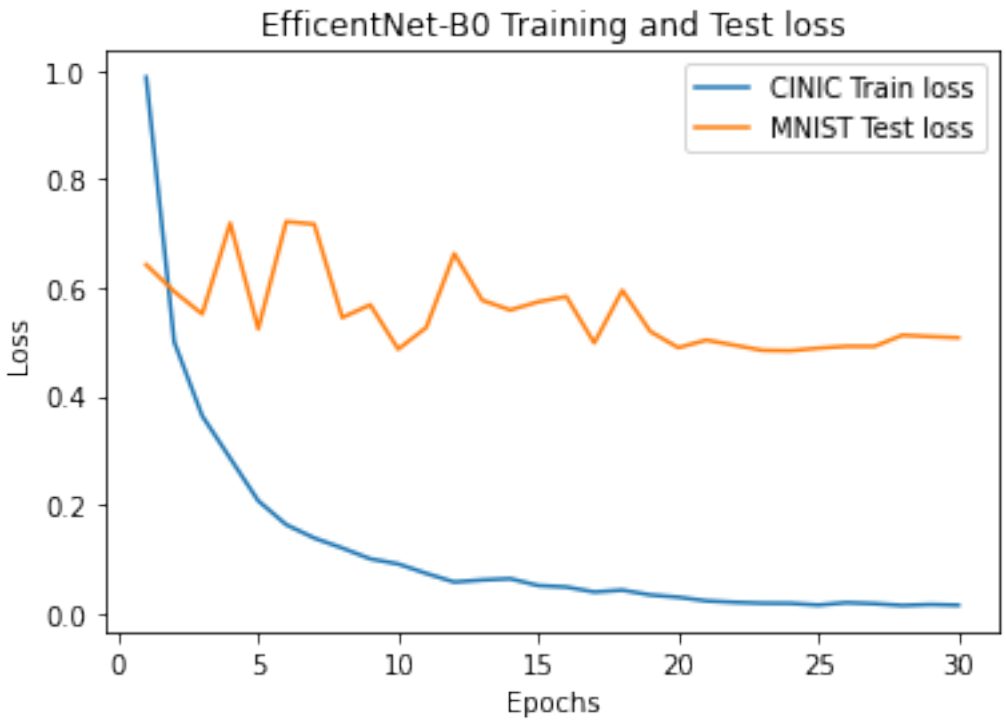
Model Evaluation with CIFAR10 test set

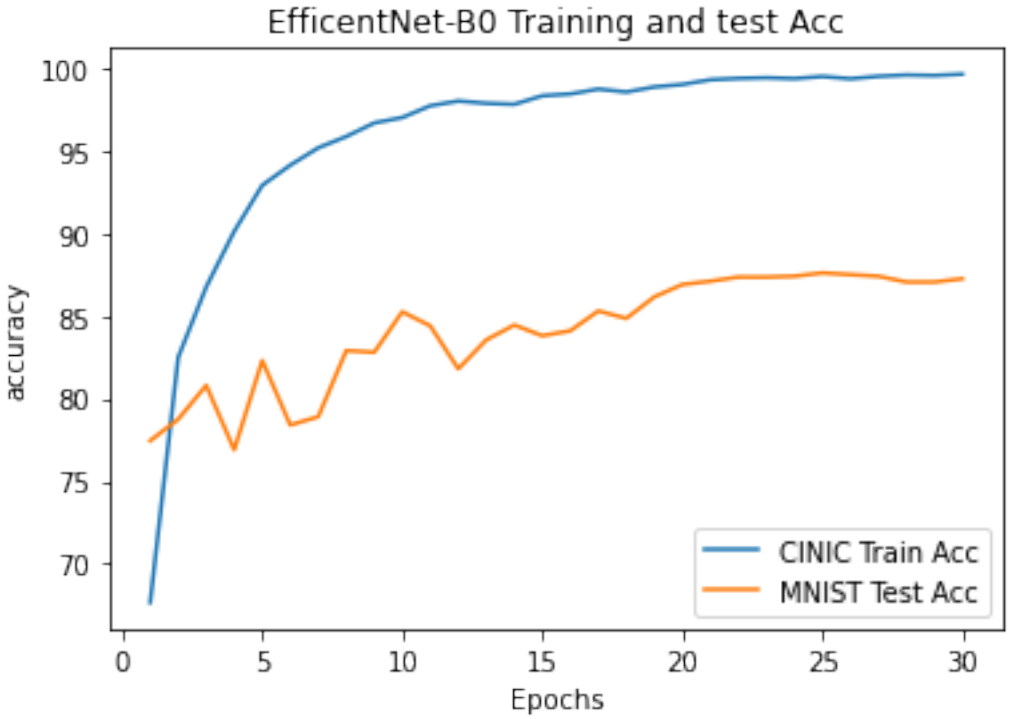
[16]:
```
model.eval()
test_loss = 0
correct = 0
total = 0
with torch.no_grad():
        for batch_i, (inputs, targets) in enumerate(dataloader_test):
```

```
            inputs, targets = inputs.to(device), targets.to(device)
            outputs = model(inputs)
            loss = criterion(outputs, targets)
            test_loss += loss.item()
            _, predicted = outputs.max(1)
            total += targets.size(0)
            correct += predicted.eq(targets).sum().item()
            if batch_i > 0 and (batch_i + 1) % 100 == 0:
                print("test-> epoch {}, iter {} / {}, Loss: {:.3f} | Acc: {:.
 ↪3f}".format(
                    epoch+1,
                    batch_i+1,
                    len(dataloader_test),
                    test_loss/(batch_i+1),
                    100.*correct/total,
                ))
acc = 100.*correct/total
print("Accuracy: {:.4f}".format(acc))
```

```
Accuracy: 87.2500
```

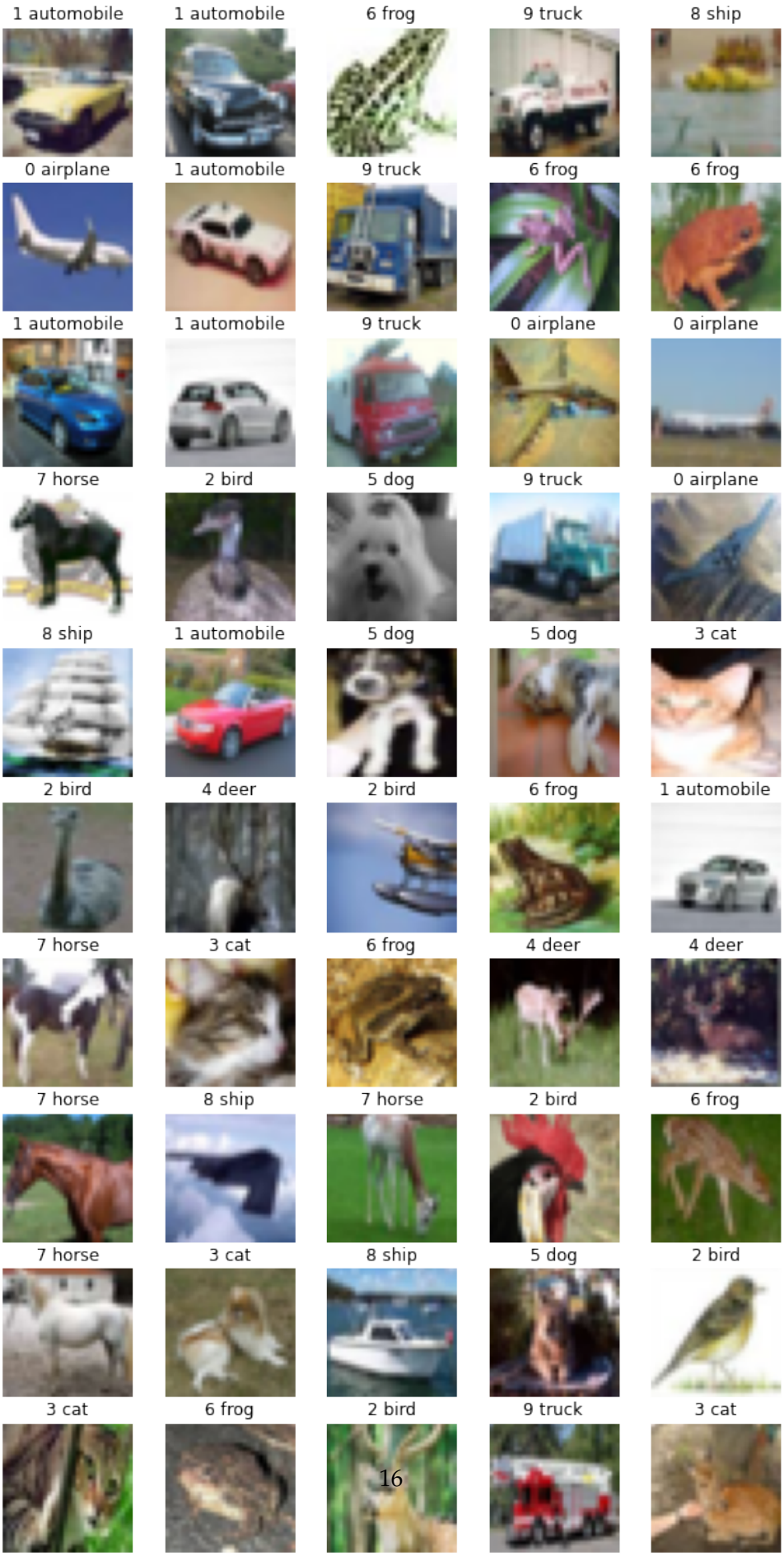
[17]:
```
# loss_train = history.history['train_loss']
# loss_val = history.history['val_loss']
loss_train=train_loss_epoch
loss_test=valid_loss_epoch
epochs = range(1,len(train_loss_epoch)+1)
plt.plot(epochs, loss_train, label='CINIC Train loss')
plt.plot(epochs, loss_test, label='MNIST Test loss')
plt.title('EfficentNet-B0 Training and Test loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()

epochs = range(1,len(valid_acc_epoch)+1)
plt.plot(epochs, train_acc_epoch, label='CINIC Train Acc')
plt.plot(epochs, valid_acc_epoch, label='MNIST Test Acc')
plt.title('EfficentNet-B0 Training and test Acc')
plt.xlabel('Epochs')
plt.ylabel('accuracy')
plt.legend()
plt.show()
```


EfficentNet-B0 Training and Test loss
CINIC Train loss
MNIST Test loss
Loss
Epochs
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20
25
30


EfficentNet-B0 Training and test Acc
CINIC Train Acc
MNIST Test Acc
accuracy
Epochs
100
95
90
85
80
75
70
0
5
10
15
20
25
30

[ ]:
```
# !unzip -q  '/content/gdrive/MyDrive/DLCourseProject/train.zip' -d '/content/
 ↪train'
# !unzip -q  '/content/gdrive/MyDrive/DL_Course/test.zip' -d '/content/test'
```

[18]:
```
from cifar10_pytorch.cifar10_pytorch.data import transform
validation_img_paths = []
transform_test = transform.build_transforms(cfg, is_train=False)
for i in range(1,51):
  validation_img_paths.append("/content/test/all/"+str(i).zfill(4)+".png")
img_list = [Image.open( img_path) for img_path in validation_img_paths]

validation_batch = torch.stack([transform_test(img).to(device)
                                for img in img_list])

pred_logits_tensor = model(validation_batch)
pred_probs = F.softmax(pred_logits_tensor, dim=1).cpu().data.numpy()
classes=['airplane','automobile','bird', 'cat','deer', 'dog', 'frog','horse',␣
 ↪'ship',  'truck']
fig, axs = plt.subplots(10, 5, figsize=(10, 20))
for i, img in enumerate(img_list):
    ax = axs[int(i/5),i%5]
    ax.axis('off')
    # print(pred_probs[i])
    ax.set_title(str(np.argmax(pred_probs[i]))+" "+classes[np.
 ↪argmax(pred_probs[i])])
    ax.imshow(img)
```


1 automobile
1 automobile
6 frog
9 truck
8 ship
0 airplane
1 automobile
9 truck
6 frog
6 frog
1 automobile
1 automobile
9 truck
0 airplane
0 airplane
7 horse
2 bird
5 dog
9 truck
0 airplane
8 ship
1 automobile
5 dog
5 dog
3 cat
2 bird
4 deer
2 bird
6 frog
1 automobile
7 horse
3 cat
6 frog
4 deer
4 deer
7 horse
8 ship
7 horse
2 bird
6 frog
7 horse
3 cat
8 ship
5 dog
2 bird
3 cat
6 frog
2 bird
9 truck
3 cat

[ ]: