BALANCED MULTI-TASK LEARNING OF ARTIST GROUP FACTORS

Kenny Falkær Olsen *s164396* *Johannes Gade Gyldenkærne* *s164397* *Daniel Thoren* *s144222*

Technical University of Denmark

ABSTRACT

Recognising musical genre from raw audio alone is a difficult problem, as genre labels are often noisy from annotators not having access to a standardised genre labelling approach. To overcome this limitation, we propose to exploit the relationship between artists and genre by training a deep multi-task network to simultaneously predict both genre and noise-free artist-related features, for more robust genre classification. A PyTorch implementation of the networks is provided.[1]

***Index Terms*—** Multi-task learning, transfer learning, adaptive loss balancing, feature extraction

1. INTRODUCTION

In this work we train deep neural networks to perform accurate classification of musical genre from raw audio on the medium size *Free Music Archive* (FMA) dataset, containing 25,000 30 second clips of music annotated with main genre, subgenres, artist labels, album labels and many other kinds of metadata in an attempt to replicate the findings of [1].

The FMA dataset is crowd-sourced, and genre labels are thus noisy from lack of a unified genre taxonomy for consistent genre labelling. This noisiness is a problem, as a neural network may struggle to learn a generalising mapping from audio to genre, and instead overfit to the noisy distribution of the dataset. To counteract this, we consider exploiting the relationship between song artist and genre.

Artist labels can be considered much more objective than genre labels, as they do not suffer from taxonomy problems. Furthermore most artists produce songs belonging to a fairly small subset of all genres. As such, we can think of individual artists as discrete probability distributions over genres, such that for every artist, each genre has a certain probability of being expressed in a song belonging to that artist.

It would thus seem reasonable that using artist information as part of predictions would strengthen the final genre prediction. However, as we want to work with only raw audio at prediction time, we instead present a multi-task transfer learning framework, where we first train a multi-task network to predict both genre *and* artist-related information, and then fit another single-task network to the high-level output features of the multi-task network. In this way, to make predictions for an audio signal, we would first make predictions using the multi-task network, and then feed the output into into single-task network, which makes a final genre prediction. This kind of network is shown in [1] to be superior to a single-task network (with the same number of parameters) targeting the genre labels directly. Thus, we focus only on improving the multi-task network in this work.

[1]`https://github.com/falkaer/artist-group-factors`

2. METHODS

Convolutional neural networks (CNNs) have recent seen use in music information retrieval [2], where a CNN is trained on (typically) spectrograms of the audio, whereby the problem becomes similar to a traditional image recognition problem. Our proposed architecture is trained on *mel spectrograms* (see fig. 1), which are standard spectrograms projected onto the *mel scale*, a scale of pitches based on human hearing.

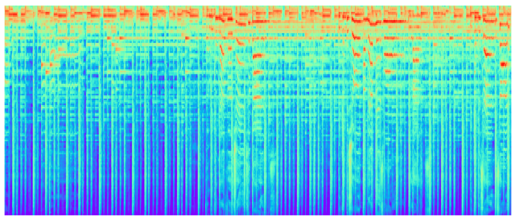

Fig. 1: A heatmap of a mel spectrogram over 30 seconds of the folk song *Overboard* by John Mease.

Targeting artists in the multi-task network *directly* is unfeasible due to data sparsity, as each artist may have only a few tracks. We therefore instead seek to target *clusters* of artists in order to avoid learning bottlenecks associated with having a large numbers of sparse classes. The following section describes a method for extracting such clusters from raw audio and artist labels first used in [1], whereby artists are clustered based on features extracted for their songs.

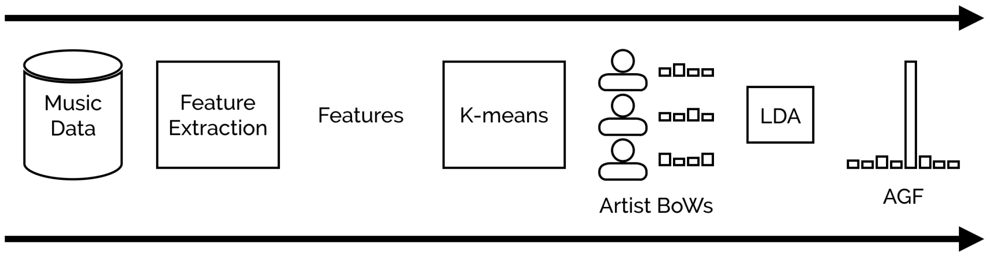

Fig. 2: The *Artist Group Factor* extraction pipeline.

2.1. Artist Group Factors

The main problem to overcome when trying to cluster artists is to come up with a suitable similarity measure between two artists, which can be used to meaningfully cluster artists together.

Artist Group Factors (AGFs) achieve this by first clustering songs by a given descriptive feature (more on that later) through the K-means algorithm into 2048 clusters, such that the features of every song now fit into exactly one of these clusters.

Then, each artist is assigned a 2048-dimensional Bag-of-Words (BoW) vector, where the i'th element of the vector is how many song features belonging to that artist are present in the i'th cluster. In a sense, the elements of this vector represent the frequency of each cluster for the specific artist.

As this vector is often very sparse, it is further dimensionality reduced to 40 dimensions using Latent Dirichlet Allocation, which models the BoW vectors as a linear combination of a set of hidden underlying topics. The resulting vector is called an *Artist Group Factor*, and is used as a target for all songs belonging to its artist during training, see section 3, *Proposed architecture* and fig. 3.

2.1.1. Descriptive features

The next problem is to select which features to cluster the songs by when creating AGFs. In training the multi-task network, we used the following low-level descriptive features for audio:

- *Mel-frequency cepstral coefficients (MFCCs)*, which are commonly used in text-to-speech and music information retreival to represent timbre of audio.[3]

- *Chromagrams*, which divide audio content into the 12 semitones of octaves commonly found in music, thereby representing harmonic content.

- *Spectral contrast*. In spectral analysis of audio, spectral peaks and valleys respectively describe abundance and lack of harmonic content. Spectral contrast thus describes the "relative distribution of the harmonic and non-harmonic components" in a musical signal.[4]

We also construct a fourth AGF from the subgenre labels given in the FMA dataset. We directly construct a 161-dimensional (the number of different subgenres in the dataset) BoW vector from counting the presence of subgenres across all songs belonging to each artist, and then fit an LDA model ontop to produce 40-dimensional AGF vectors.

3. PROPOSED ARCHITECTURE

The multi-task network consists of a total of seven convolutional layers and two fully connected layers, with the first convolutional layer being shared across all tasks. This is to allow the first layer of the network to learn a "pre-processing" function common to all layers (saving on training time), and may also have a regularising effect as the first layer is then forced to generalise across several slightly different tasks. This does however require us to ensure that all tasks have equal influence over the shared layer, which we address in section 4.1 *Training the multi-task network*.

After the seven convolutional layers, the output is 256 filters whose dimensions depend on the dimensions of the input spectrogram. To each of these 256 filter outputs we apply *global average pooling*, where we take the average across all activations in the filter output. The output of this operation is a 256-dimensional vector. Intuitively, this can be thought of as an operation that measures how strongly each filter is activated, and assuming that the filter has learned to recognise a certain feature, how strongly that feature is represented in the input.

These 256-dimensional vectors are then connected to two fully connected layers, and softmax is applied to the output layer for predictions.

In the single-task network, the input is the 256-dimensional vectors from the global average pooling layer of the multi-task network, disregarding the fully connected layers in the multi-task network. The single-task network then consists of a single fully-connected layer which also ends in a softmax activation for prediction.

See the implementation reference on GitHub for individual layer parameters.

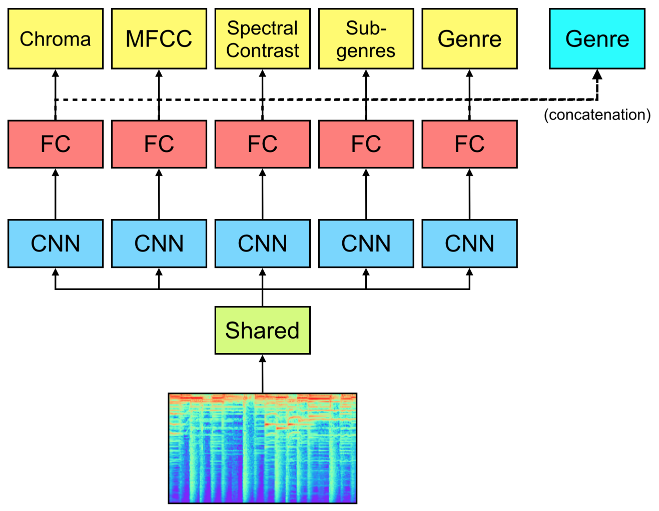

Fig. 3: The proposed architecture takes a mel spectrogram frame (see section 4.3.1 *Data augmentation*), and targets each of the yellow targets as part of the multi-task network. The single-task network then later uses outputs from the multi-task CNN layers to make a final genre prediction.

4. TRAINING

We train our networks in two phases; the first phase is training a multi-task network targeting 4 different AGFs as well as a fifth genre target, and the second phase is fitting a single-task network on top of the CNN outputs from the multi-task network.

We train all networks through standard back-propagation using the Adam optimiser with a fixed learning rate of 0.0001 and a batch size of 64 (frames, see 4.3.1 *Data augmentation*) across all experiments. In both training phases we employ a 85/15% training/validation split. We trained all networks on 2x NVIDIA Tesla V100 16GB PCIe GPUs, which took about 18 hours per run for the multi-task network and 3 hours for the final genre predictor.

4.1. Training the multi-task network

Across all tasks the individual loss functions $L_i(t)$ are cross-entropy loss. For the genre target we use the built-in PyTorch cross-entropy loss function, which targets one out of C classes with probability 1.

For AGF targets which consist of 40 dimensional vectors, we take the softmax of each AGF and target the resulting probability distribution with cross-entropy loss. Targeting a probability distribution is not implemented in PyTorch by default, so we implement it ourselves as

$$CE(y, \hat{y}) = -\sum p(y) \log q(\hat{y})$$

where $p(y)$ is the target (softmaxed AGF) distribution, and $q(\hat{y})$ is the networks predicted distribution (logits with softmax).

4.1.1. Loss balancing

In the first phase the main problem is how to compute a total loss term from the individual task losses $L_i(t)$, such that each task has equal influence over the gradients of the shared layers, W. To do this, we use *GradNorm* [5], an algorithm that adaptively learns weights $w_i(t)$ at each timestep t for each task as part of back-propagation such that the total loss is $L(t) = \sum w_i(t) L_i(t)$.

GradNorm aims to ensure that both gradient magnitudes and training rates (rates at which the losses change) at the shared layer W stay as close to uniform across all tasks as possible. The following definitions are used in *GradNorm*:

- W: the parameters of the shared layer in the network.
- $G_W^{(i)}(t) = \|\nabla_W w_i(t) L_i(t)\|_2$: the L_2 norm of the gradient of the weighted task loss w.r.t. W.
- $\bar{G}_W(t) = \mathrm{E}[G_W^{(i)}(t)]$: the average gradient norm across all tasks.
- $\tilde{L}_i(t) = L_i(t)/L_i(0)$: the loss ratio or inverse training rate for each task, where lower values mean the task is learning faster than higher values. $L_i(0)$ is the task loss at timestep $t = 0$.
- $r_i(t) = \tilde{L}_i(t)/\mathrm{E}[\tilde{L}_i(t)]$: the relative inverse training rate of each task, where values above 1 means the task is learning faster than the average task.

The goal of *GradNorm* is then to adjust the weights $w_i(t)$ such that each tasks gradient norm is pushed closer to the average gradient norm, while also factoring in the tasks relative training rate. This is done by targeting

$$G_W^{(i)}(t) \mapsto \bar{G}_W(t) \times [r_i(t)]^{\alpha}$$

where the hyperparameter α controls the asymmetry between pushing the task's gradient norm towards the average gradient norm and rate-limiting the task by the relative inverse training rate. That is, for $\alpha = 0$, the algorithm will try to pin all gradient norms to the average gradient norm, and for larger values of α, the algorithm will favour normalising the training rates of individual tasks. Tuning this hyperparameter leads to performance benefits, but gives similarly positive results for several values of α between 0 and 2, and in all cases gives better performance than not using *GradNorm* (aka. with $w_i(t) = 1 \, \forall \, t$). We found that $\alpha = 0.5$ works best with the FMA dataset and proposed network.

4.1.2. Updating $w_i(t)$

In order to update $w_i(t)$, the *GradNorm* loss is defined as the sum of L_1 distances between the current task gradient norms and the target gradient norm,

$$L_{grad} = \sum_i \left| G_W^{(i)}(t) - \bar{G}_W(t) \times [r_i(t)]^\alpha \right|_1$$

and the gradients of this loss are then taken w.r.t. each of the weights $w_i(t)$, and used to update the weights $w_i(t+1)$ (while keeping the target $\bar{G}_W(t) \times [r_i(t)]^\alpha$ constant). For updating the gradient weights we use the Adam optimiser with a learning rate of 0.001. After every update, the weights are renormalised such that $\sum w_i(t) = T$, where T is the number of tasks, to prevent the optimiser from driving the weights to zero.

4.1.3. Initialising $L_i(0)$

Since the training rates depend strongly on the initial task losses $L_i(0)$, unstable initialisations can lead to very different results using this loss balancing method. Since our network trains quite slowly and the training loss fluctuates a lot, it is not unlikely for $L_i(0)$ to be set to a task loss well above or below the average task loss for many of the following timesteps t, causing the weight $w_i(t)$ to be forced out of optimal ranges.

In our experiments, fixing $L_i(0)$ to the initial loss at the very start of training (timestep 0 of epoch 0) gives inconsistent performance that is highly dependant on the random initialisations of the network weights. Setting $L_i(0)$ to the first loss in each epoch allowed the algorithm to better reflect the actual training rates throughout the run, but sometimes lead to large fluctuations in weights from epoch to epoch, as the initial loss could still easily be much higher or lower than the average loss throughout the epoch.

The original *GradNorm* paper also proposed to initialise $L_i(0)$ as $L_i(0) = \log C_i$ where C_i is the number of classes in task i for a classification problem using cross-entropy. Since all our tasks use cross-entropy for their loss functions $L_i(t)$, we were able to use this initialisation, which consistently gave much smoother training curves than the other initialisations, appearing similar to the best training runs of the per-epoch initialisations, except that the weights were much slower to change, as this one-shot form of initialisation of $L_i(0)$ cannot reflect local changes in training rate. This kind of initialisation gave somewhat lower performance than the per-epoch initialisation.

Alternatively, initialising $L_i(0)$ to a moving average over the task losses at the start of every epoch could perhaps have been a more robust version of the simpler per-epoch approach, while being more adaptable to local changes in the training rates than the class-log initialisation.

4.2. Training the single-task predictor

Finally, we train a single-task network consisting of a single fully connected layer on the outputs of the CNN blocks of the multi-task network. This network targets genre directly, also using the standard PyTorch implementation of cross-entropy loss.

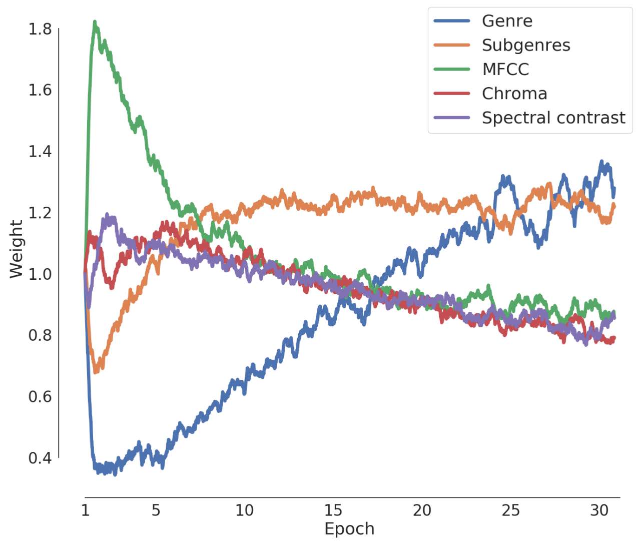

Fig. 4: The weights $w_i(t)$ across a 30 epoch training run with per-epoch $L_i(0)$ initialisation with only minor fluctuations. The genre weight is initially suppressed to below 0.4, but later overtakes other tasks in weight, suggesting that the genre target is easily learned to a certain point, but then becomes harder than other tasks which train more steadily.

4.3. Preventing overfitting

We use dropout and batch normalisation sparsely across both networks, and also use L_2 regularisation at training time. That is, we add $\lambda \|\mathbf{W}\|_2^2$ where $\lambda = 0.00001$ to the total loss during training to keep the network weights from growing too large.

As the *FMA* dataset is highly imbalanced (the most frequent class has 7097 observations, and the least represented has only 21 observations), when training networks targeting genre directly, we weigh the loss from observations belonging to each class by the inverse of the class frequency. In this way, observations belonging to infrequent classes contribute more to the total loss than frequently occurring classes.

4.3.1. Data augmentation

We split the mel spectrograms that the network takes as input into 7 frames with 50% overlap, effectively increasing the amount of observations by 7 times. This leads to improvements across all performance metrics, and greatly improves mean F1 score, as it allows the network to learn how to make predictions for low frequency classes better.

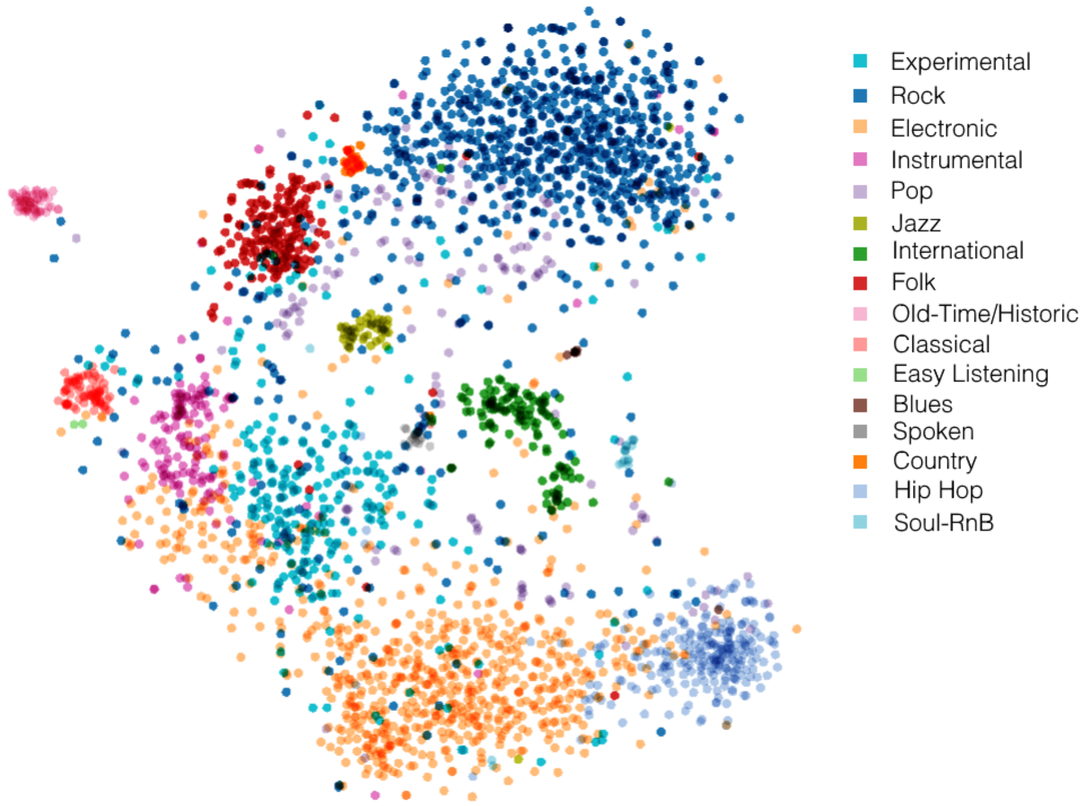

Fig. 5: The 1024-dimensional logits of the last layer in the final genre prediction network for all observations in the dataset, reduced to 2 dimensions with t-SNE for visualization. While most classes are nicely clustered and can therefore be expected to have reasonably accurate predictions, the model appears to struggle the most with learning the pop (light purple) genre, and also has some trouble with the experimental genre. This aligns well with our intuition, as both the pop and experimental genres are very loosely defined.

5. RESULTS & DISCUSSION

The final model is validated to have a log loss of **0.362**, a total (micro) F1 score of **87.99%** and a mean (macro) F1 score of **91,08%**. It remains to be seen how well the model generalises however, as it is no longer possible to test our model against the test data from the challenge the original authors of [1] participated in.

Looking at fig. 5 where we visualize the activations in the networks last layer for every observation in the dataset, we see that the model learns to cluster most classes well. The model struggles with some classes, such as pop and experimental in an intuitively expectable way (pop and experimental are both very loosely defined), but classes which are extremely underrepresented (easy listening with only 21 observations, and blues with 74) are clustered extremely densely (though it is hard to tell from the 2D t-SNE plot, it is easy to tell on an interactive 3D plot). This leads us to suspect that the model is overfitting these very underrepresented classes strongly, as it is too difficult for the network to learn anything general about those classes with so few observations.

6. REFERENCES

[1] Jaehun Kim, Minz Won, Xavier Serra, and Cynthia C. S. Liem, "Transfer Learning of Artist Group Factors to Musical Genre Classification," *arXiv e-prints*, p. arXiv:1805.02043, May 2018.

[2] Monika Doerfler and Thomas Grill, "Inside the spectrogram: Convolutional neural networks in audio processing.," July 2017.

[3] Steve Tjoa, "Mel frequency cepstral coefficients (mfccs)," .

[4] Dan-Ning Jiang, Lie Lu, HongJiang Zhang, Jianhua Tao, and Lianhong Cai, "Music type classification by spectral contrast feature," in *ICME*, 2002.

[5] Zhao Chen, Vijay Badrinarayanan, Chen-Yu Lee, and Andrew Rabinovich, "GradNorm: Gradient Normalization for Adaptive Loss Balancing in Deep Multitask Networks," *arXiv e-prints*, p. arXiv:1711.02257, Nov. 2017.

Appendix

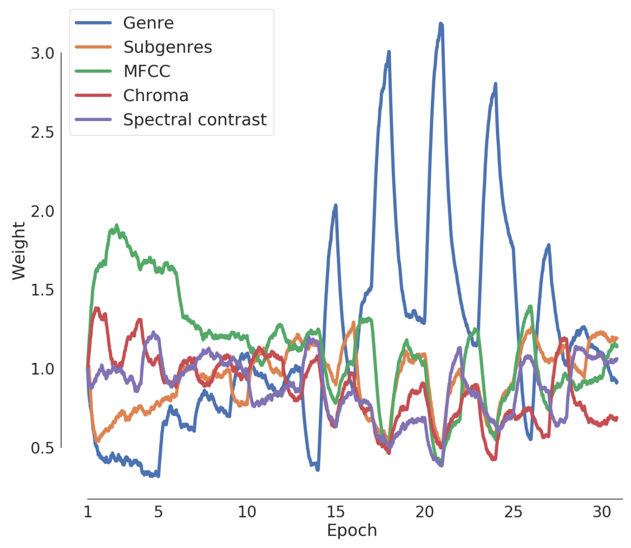

Fig. 6: The weights $w_i(t)$ across 30 epochs with heavy fluctuations due to poorly conditioned per-epoch $L_i(0)$ initialisation. The learning curves start off similar to fig. 4, but then begin to fluctuate strongly when the training appears to hit a plateau in the loss function.

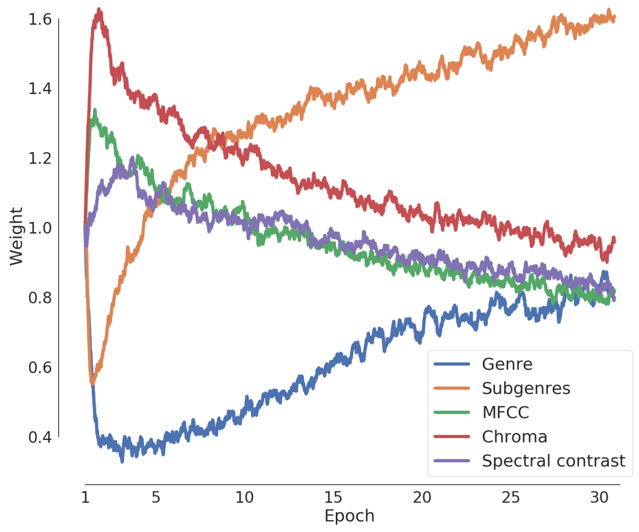

Fig. 7: The weights $w_i(t)$ across 30 epochs with no fluctuations due to one-shot $L_i(0) = \log C_i$ initialisation. The learning curves are similar to fig. 4, except that the genre weight goes up much more slowly, leading to degraded training performance.