HW1
Classification Task on Cifar-10

Bc. Martin Vozár

1 Approach

The assignment proposes working with a standardized dataset on a straight-forward task and encourages trying multiples of different configurations. First step in the process should be finding and choosing appropriate configuration as a baseline.

For those purposes, we developed a handy framework for setup of various configurations, similar to hyperparameter optimization (see *configs/** for more details). We are plotting torch.nn.CrossEntropyLoss as our loss function, and accuracy as our metric for both Training and Test set. Plots are made on a log/log scale, as it allows easier interpretation and detections of potential issues (see *log2png.py* for more details).

After finding the baseline, we further explore chosen configuration in effort to maximize accuracy.

The code was ran in *conda* environment on *Ubuntu 22.04* using a single *NVIDIA GTX 1070 Ti*.

2 Preliminary exploration

In this stage, we are mostly comparing different optimizers (SGD, SGD with Momentum=0.96, Adam, and AdamW) with different learning rates. We are using defaultvalue for batch size (128), and minimal data augmentation (RandomFlipVertical(p=0.5), RandomFlipHorizontal(p=0.5)). We are using LeakyReLU as the default activation function.

During optimization we perform gradient clipping and use weight decay argument for all optimizers as means of regularization.

For each plot, we plot horizontal lines for maximum Test Accuracy and minimum Test Loss in the respective plots.

2.1 Naive - FFCN

As a first choice meant to calibrate the general setup of different optimizers, as well as gaining intuition for necessary complexity of further examined Neural Networks, we tested a simple FCNN (further as a nickname - Naive). For this Network we used nn.Dropout(p=0.3).

Iteratively, we tuned individual learning rates for each optimizer (as their behaviour varied quite a bit) as well as the Network width and depth.

Plotted are results for a Network:

- nn.Linear(3072, 256)
- (nn.Linear(256, 256)) * 8
- nn.Linear(256, num_classes)

with activation function after each but last layer. Torch implementation of CrossEntropyLoss allows us to omit applying Softmax on output layer, as well as one-hot encoding on the labels.

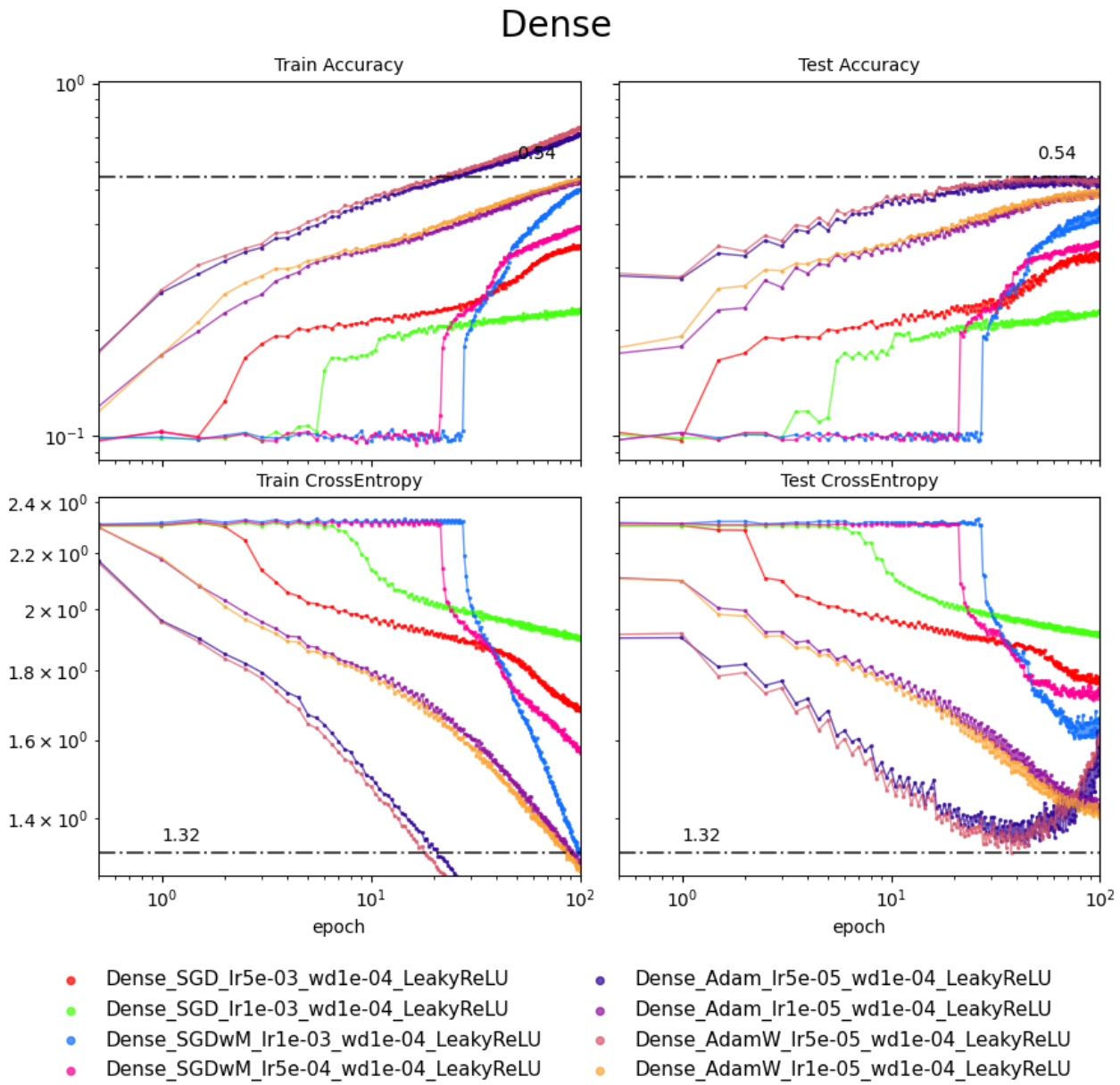

Figure 1: Plot of results of the preliminary exploration of FCNN architecture.

In those results, we clearly observe examples of overfitting in most of the runs.

The second thing we can observe is a stagnation of SGD optimization. Somewhat surprisingly, variants using Momentum stagnated for longer period of time before proceeding to overfit. However, this could be acredited to other factors, e.g. specific initialization of weights.

2.2 Convolutional Encoder

We tested a few different variants of Convolutional Encoder. All variants consisted of blocks of (nn.Conv2d layers, nn.MaxPool2d) and a final nn.Linear layer at the end. Variables were strides of the convolutions and number of blocks.

Plotted is the variant which achieved the target accuracy of 70% using SGD with Momentum as optimizer. We used this variant in the following architectures as the final encoding component.

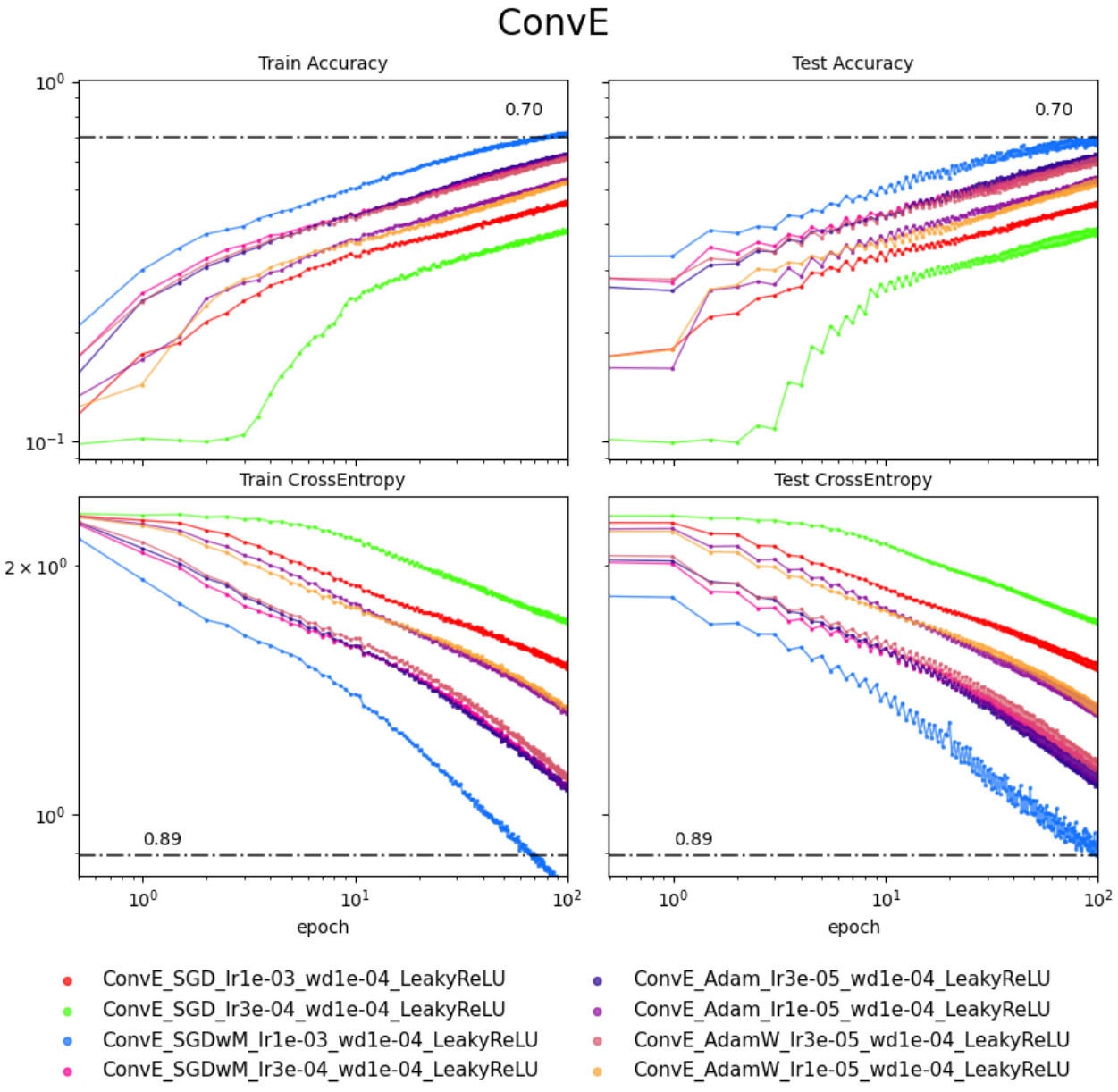

Figure 2: Plot of results of the preliminary exploration of ConvEncoder architecture.

2.3 Residual Convolutional Encoder

Drawing inspiration from ResNet architecture, we implemented a basic adaptation of the principle.

The basic idea of the architecture:

- nn.Conv2d(in_channel=3, out_channels=16, kernel_size=1, stride=1)
- N x ResBlock
- ConvEncoder

The first layer expands the number of channels to a set number, which then remains unchanged passing through the ResBlocks. Finally, Convolutional Encoder (adapted to also handle in_channels different from original images) outputs the logits.

From multiple tried and tested variants of ResBlock, we settled on definition:

- input x
- z = nn.Conv2d(in_channels=16, out_channels=32)(x)
- z = nn.BatchNorm2d(num_features=32)(z)
- z = activation(z)
- z = nn.Conv2d(in_channels=32, out_channels=16)(z)
- z = nn.BatchNowm2d(num_features=16)(z)
- output x = x + z

It has been argued (source: some internet forum) that using Dropout and BatchNorm at the same time can lead to issues during the training. However, we have not encountered (or at least identified) any artifacts, and BatchNorm can boost convergene and regularization.

Various number of in/out_channels were tested and those values were chosed as a reasonable compormise between convergence speed and potential for accuracy. We used kernel_size=3, stride=1, padding=1 bias=False in both convolutional layers.

For this configuration, we first tested for N=4 (# of ResBlocks) to find the best performing configuration. Then, we varied the depth of the Network, as well as other parameters.

In Figure 3., we observe an increase to accuracy 77%. The most accurate Network seems to have already plateaued, signifying approaching limit of this configuration.

Interpreting the behaviour of the optimizers, we select SGD with Momentum and AdamW for further examination. We observe similar behvaiour as with ConvEncoder. To explore further, we vary the num_blocks in the RN architecture, and vary learning rates and weight decay with selected optimizers.

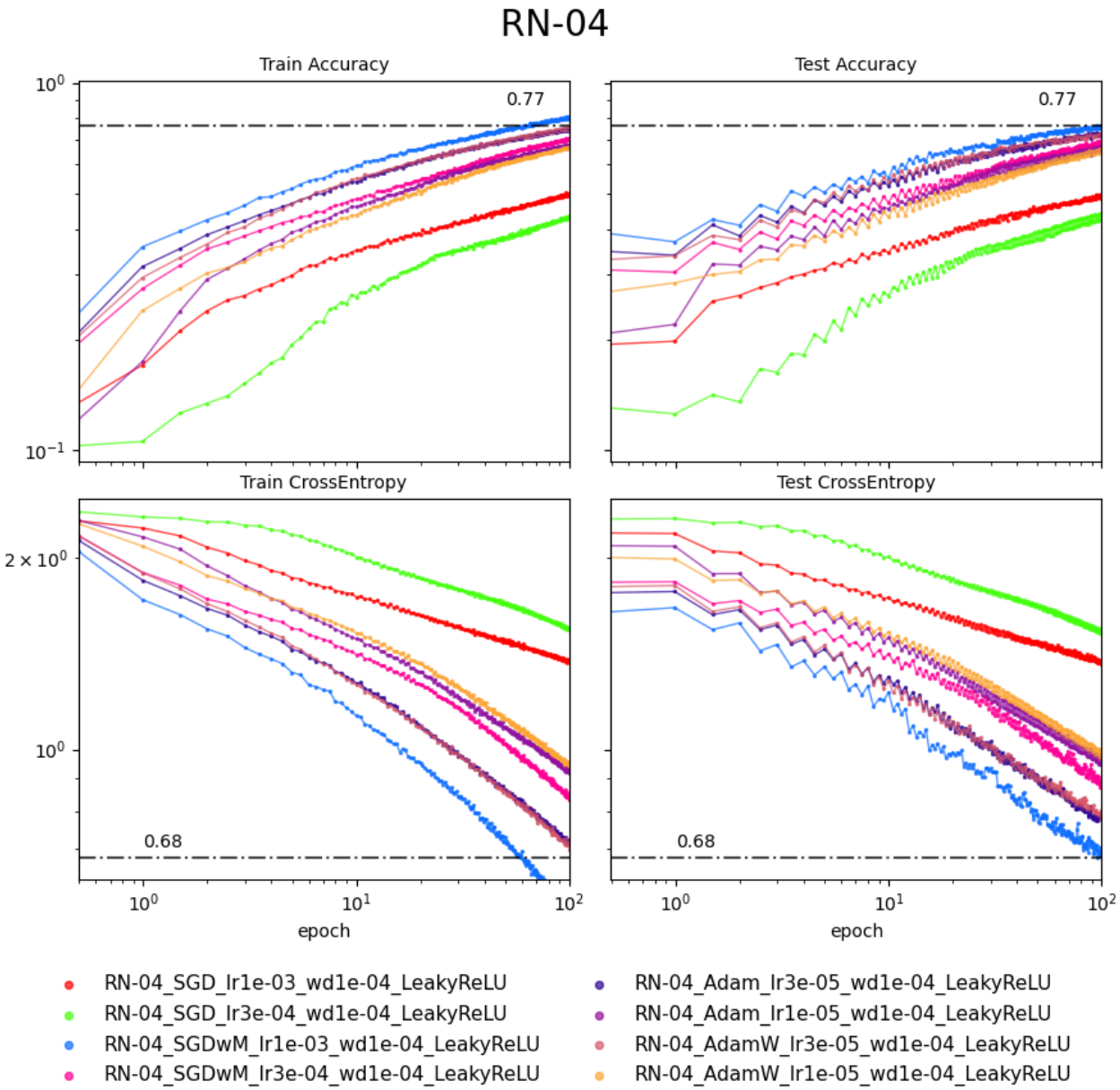

Figure 3: Plot of results of the preliminary exploration of RN04 with ConvEncoder architecture.

We achieved further improvement of Test accuraccy. However, with the individual runs getting more and more similar encouraged us to change plots to linear scale.

Results from comparison of weight_decay values give a slight hint of improved regularization.

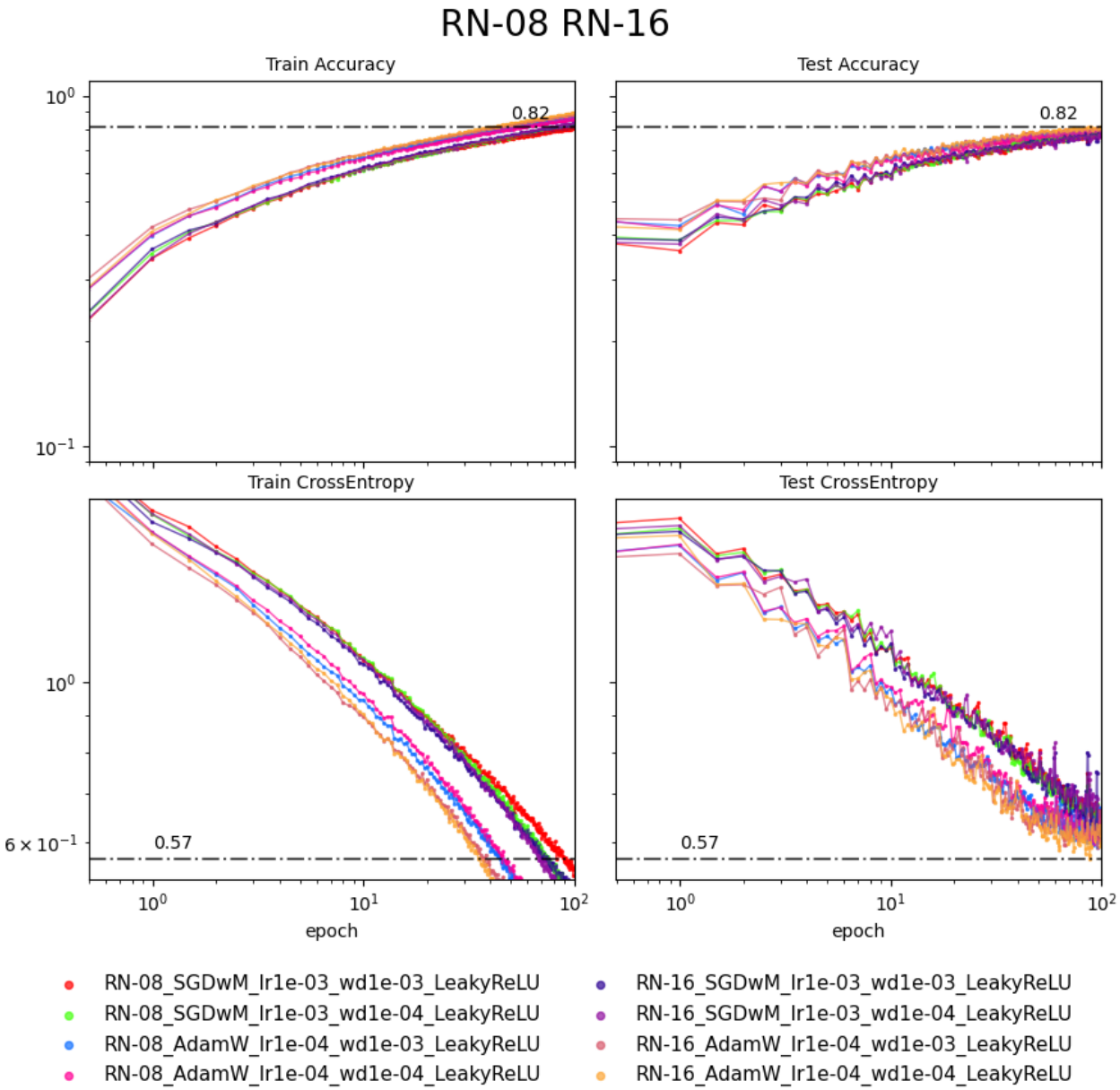

Figure 4: Closer examination of RN architecture variants.

In the following run, we also examined Tanh and Sigmoid together with LeakyReLU with the same set of configurations. We further examined different weight_decay values with expectation of better regularization. This was not observed. We prematurely stopped the run after 50 epochs, as we identified overfitting even in worse performing configurations.

For a few exploratory runs, we decided to stick to RN16, as for reducing number of efficient LR values. We also compared Sigmoid, with not much success ().

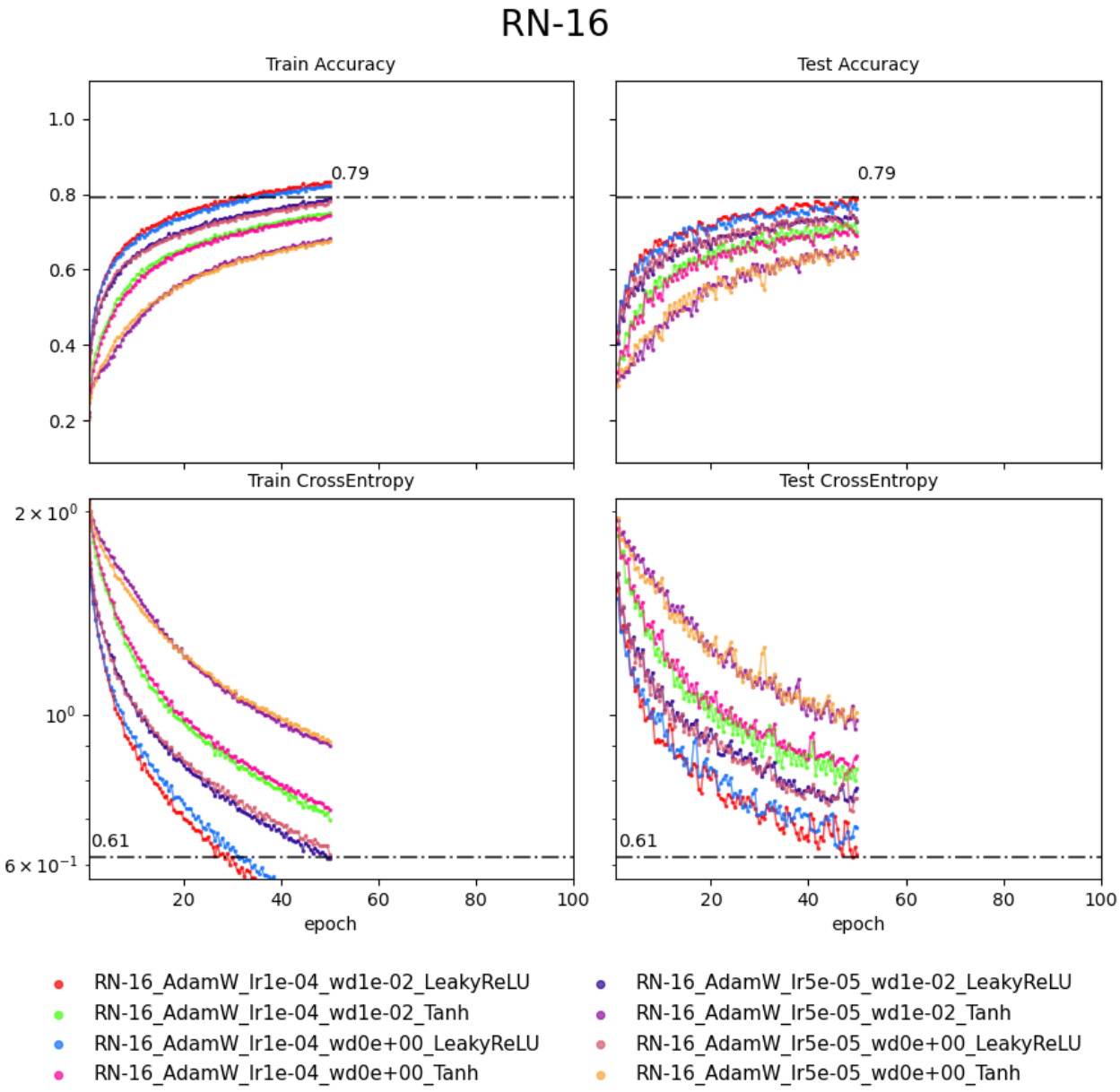

Figure 5: Even closer examination of RN architecture variants.

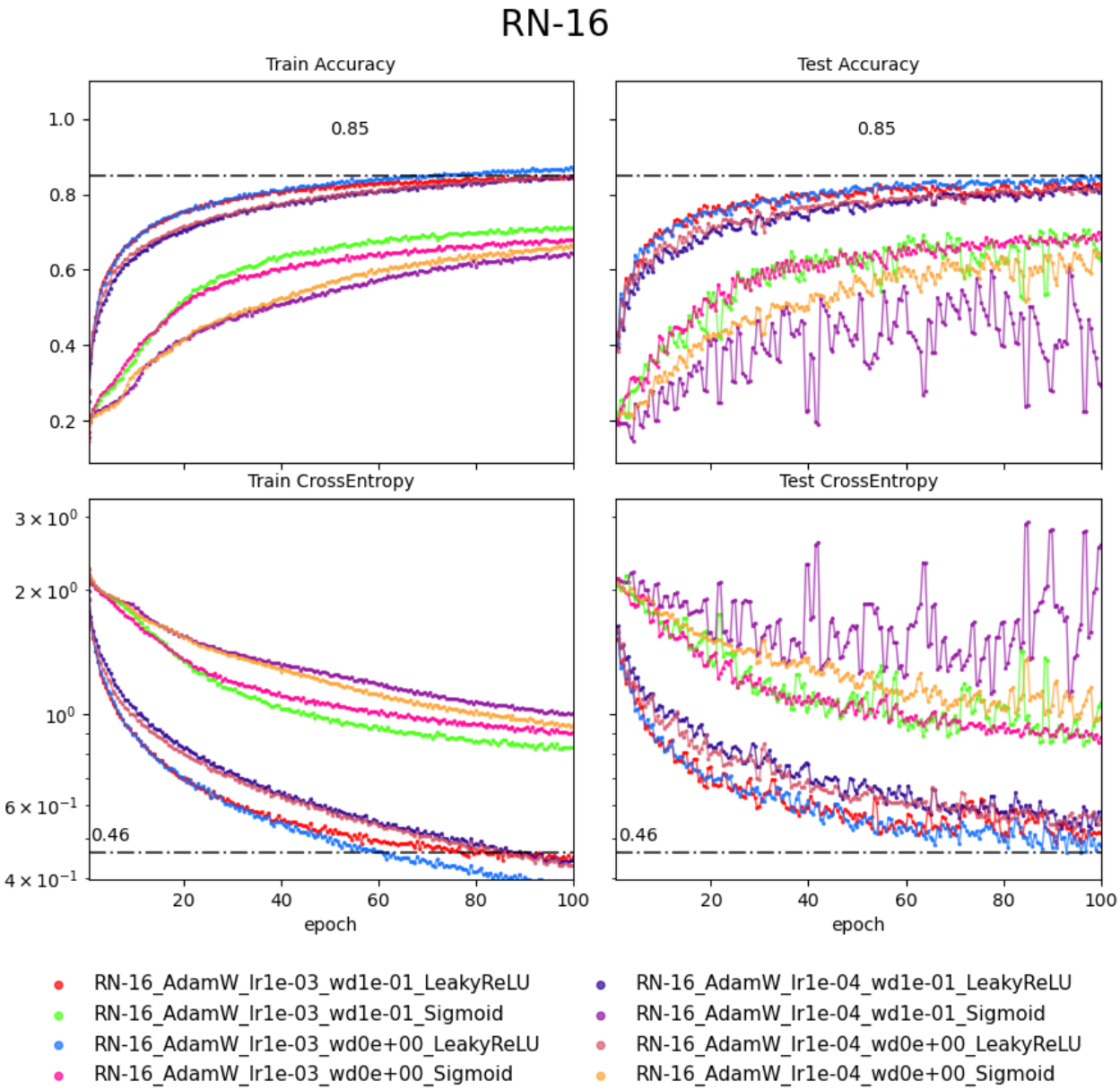

Figure 6: Examination of Sigmoid activation performance against LeakyReLU baseline.

3 Regulization and data augmentation

It should be mentioned, that with Vision Transformers scoring 99.5% accuracy on Cifar-10 benchmark one could try using Transformers. As of now, the CNNs do not seem to have reached their limist, yet, so we will stick to them.

One of the goals is to be able to stick to small-sized models, for speed and general practicality. Most effort is improving the regularization of the optimization.

We implemented a heavy_regularization option for data, applying more transformations on the inputs. weight_regularization does not seem to have the desired effect, which has not deterred s from exploring it further. We also compared various varians of Linear Function, with relatively high weight decay value. We observe less overfitting, but also a slight. Our explanation is the high value of weight_decay.

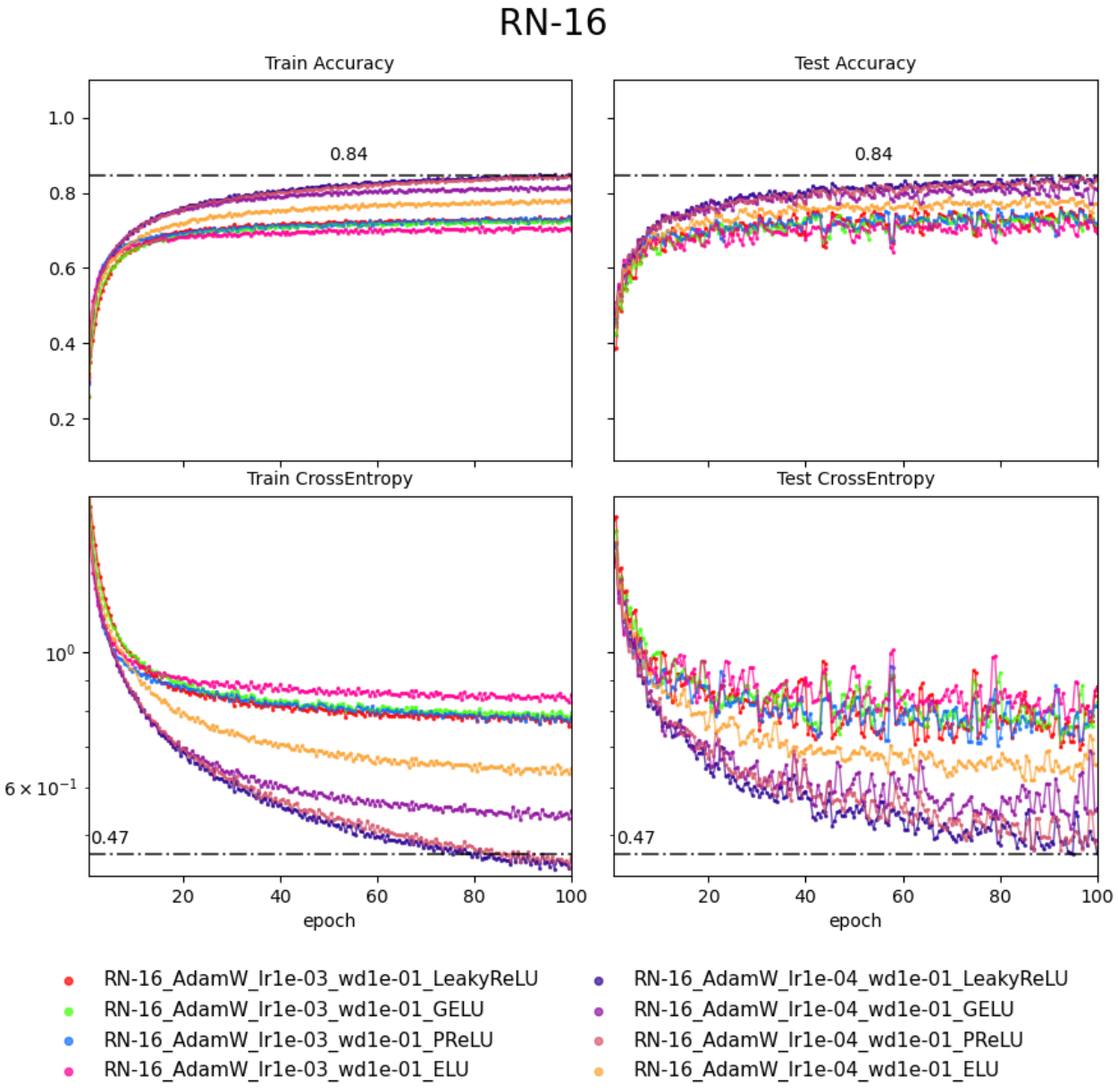

Figure 7: Examination of variations of LU variants performance against LeakyReLU baseline.

In the Figure 7. run, we also tested changing batch_size. The value we chose was 16, just to examine the effect of lower values. The effect we ascribed to this was perhaps a bit more unstable validation loss.

It is difficult to pinpoint the exact cause of individual effects. Further, we suspect a small batch_size can cause issues with heavy data augmentation. We will keep it high, as to take gradient from more samples with more representation of various augmentations. Another thing realized at this stage is mistakenly using torch.nn.Dropout where torch.nn.Dropout2d was supposed to be used.

For the following run, we included nn.Dropout2d(p=1/3) Further, based on a suspicion that heavy masking might be an issue for convolutional layers, we started thinking about a more sophisticated design with more skip connections.

The idea is, that all blocks receive the original input stacked as extra channels. Though we wanted to keep the architecture as similar to the tested RN16 variant, some adaptations were made, as to accomodate the extra input channels. The extra input channels were ignored on the residual operation: *(x = x[:, :16] + z)*.

To see, if we were going in the right direction, we tested both RN16 from previous run against the new architecture (nicknamed RR16) on the heavily augmented dataset.

3.1 Pre-training with masks

In the following runs, we iteratively selected random masking pattern with 4x4 granularity and masking proportion $p = 0.5$, $p = 0.7$, $p = 0.9$ variants. We implented them using using *torchvision.transforms.v2.Lambda*. The masks randomly zero out a portion of the image in all channels (similar to BlockDropout mechanism).

The idea was to partially reproduce approach of training Masked AutoEncoders. To reduce training time, as well as to stay focused on the task, we did not implement the Decoder, and remained using only the Encoder as previously. This is a very crude adaptation of the some parts of the principle, mostly tested out of curiosity.

We also continued optimizing the learning rate and weight decay values. As the tested architectures are almost identical, we continued using the values found to be efficient for RN also for the RR adaptation. In one of the runs, we also compared RR16 to RR32.

Though we see a decline in the accuracy, we consider this a step in the right direction. It seems that at this stage we might have over-regularized. More importatnly, we narrowed down the range of efficient learning rate values.

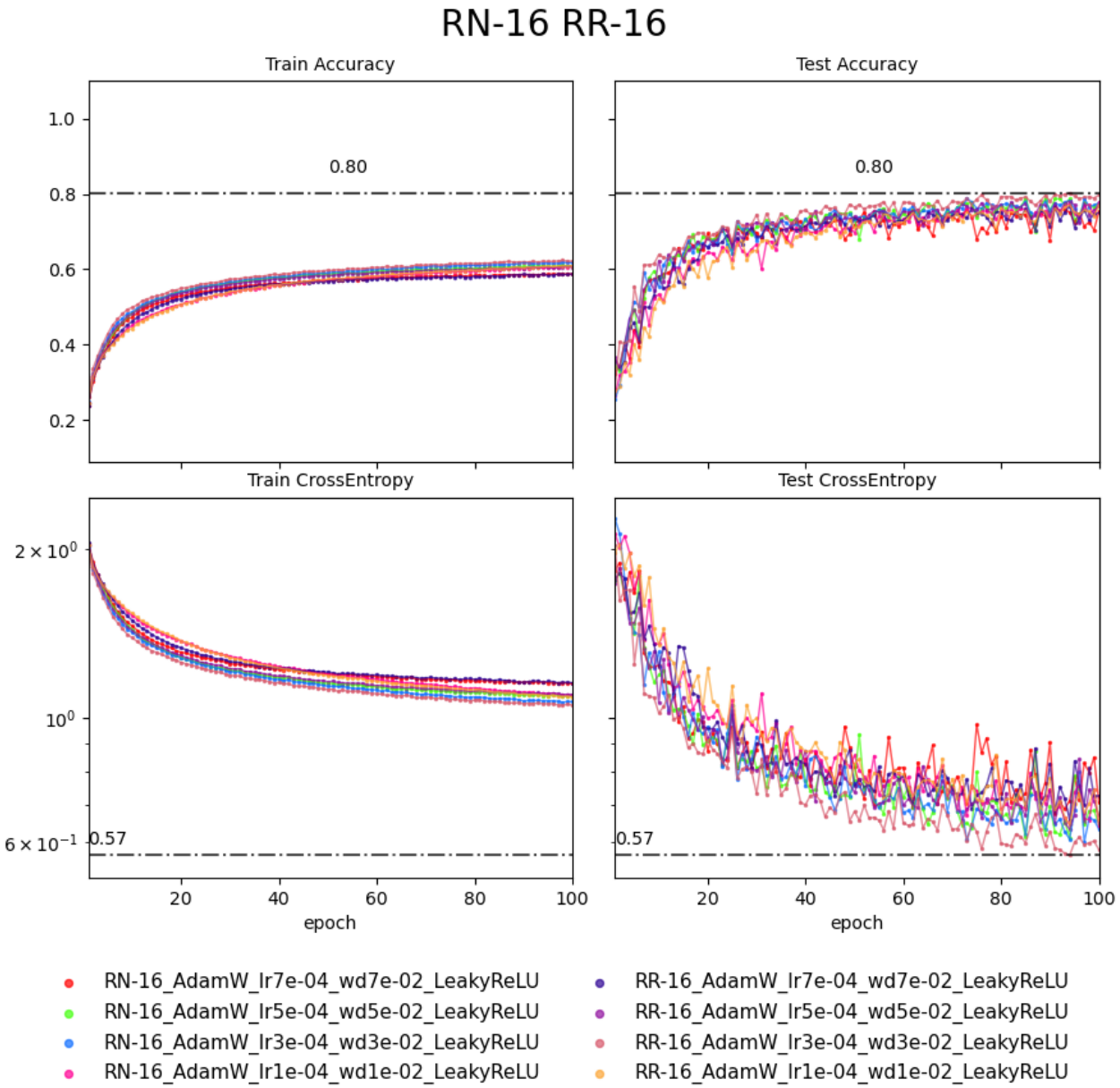

Figure 8: Comparison of RN and RR variants, comparison of narrower range of learning rate and weight decay values.

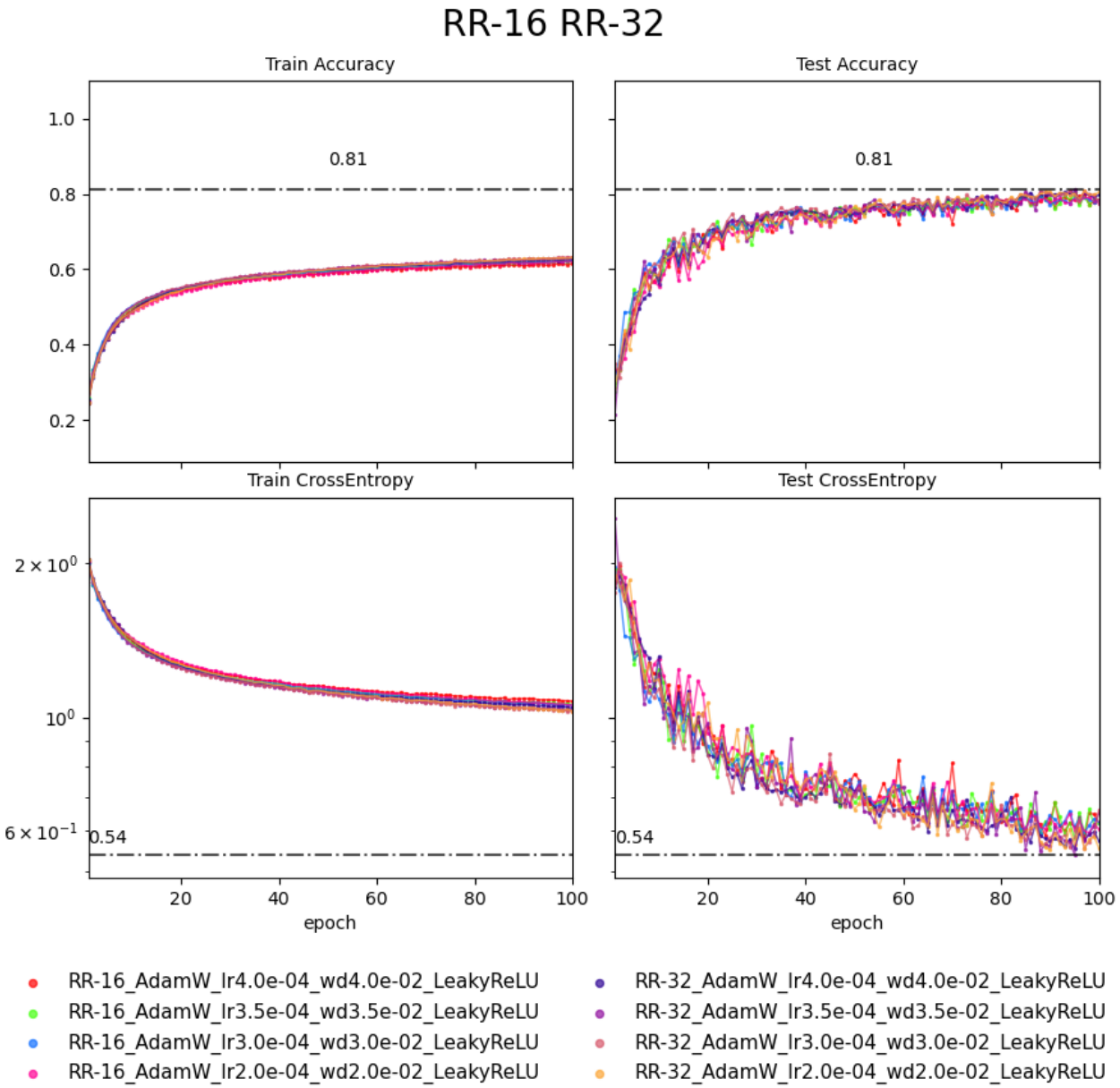

Figure 9: Comparison of RR16 and RR32 variants in a narrower range of learning rate and weight decay values.

3.2 Fine-Tuning

In the following runs, we train the best performing RR32 configuration (in terms of accuracy), together with the same configuration modified to have no weight decay.

In the first stage, we will train the model on heavily masked inputs, using $p = 0.9$, with various granularity, $4x4$, $2x2$, $1x1$ blocks. (Using the $1x1$ block is comparable to nn.Dropout, with the difference of all the channels having the same pixels blocked).

After 100 epochs, we further train with RandomCrop augmentation. We tested CosineAnnealing and CosineAnnealingWithWarmRestarts schedulers, with two 100 epoch trainings. We only show the later of those, as the main outcome is unsatisfactory.

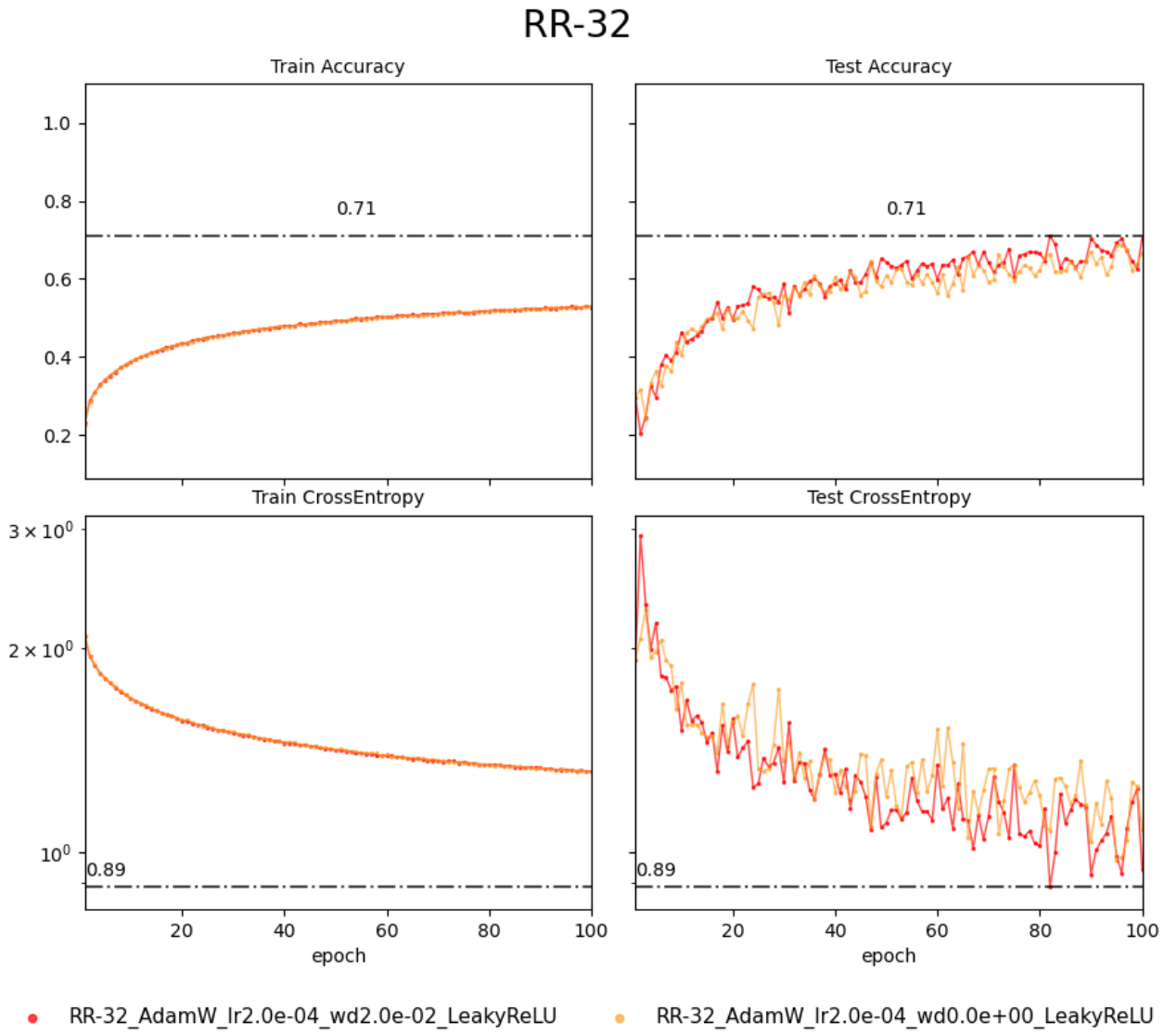

Figure 10: Training on $p = 0.9$ mixed granularity masking.

It might be still worth mentioning, that the RR architecture managed to get 71% accuracy 1 training on random 10% of the input images. Tha accuracy there was lower, which we might call an overregularization.

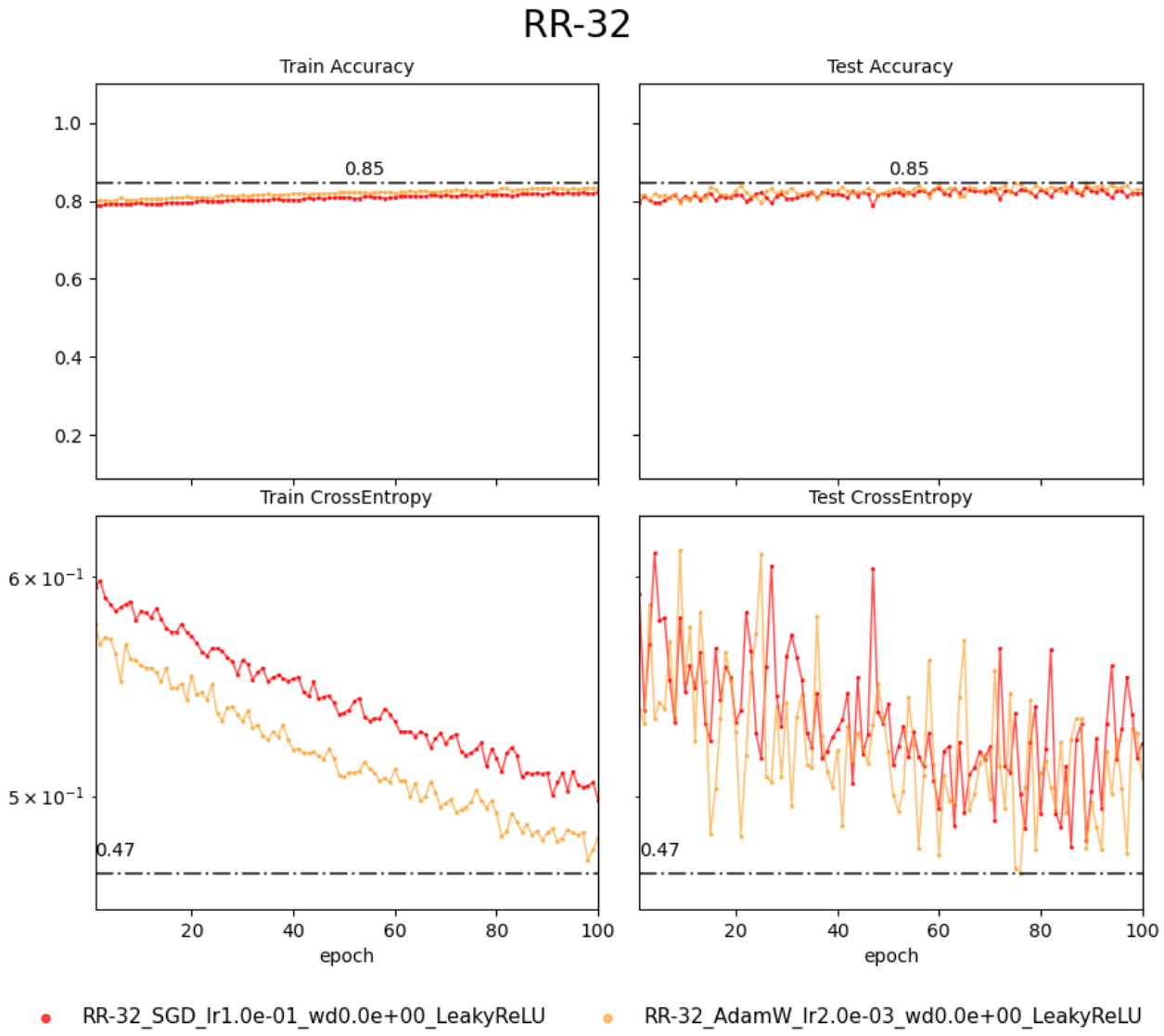

Figure 11: Second 100 epoch finetuning run.

The performance only caught up to some of the previous runs, even after taking three times as long.

4 Repository Reproduction

We managed to find a repo `https://github.com/akamaster/pytorch_resnet_cifar10` claiming to have reproduced the original ResNet experiment.

We took the model from the repository and tried adapting it to fit in the default initialization method (RC). We also altered the data loading routine to in accord with the repository, and included MultiStepLR scheduler. We also implemented a variation of the adapapted architecury, switching order of BatchNorm and Con2d operations (RCi).

Though we attempted to copy the architecture completely, it performed different to when it was

initialized from the original code (RCog). It was not examined further, so it might have been an initialization specific issue.

Even though the model from the repository had the best results, we failed to reproduce their results, hopefully due to a mundane error.

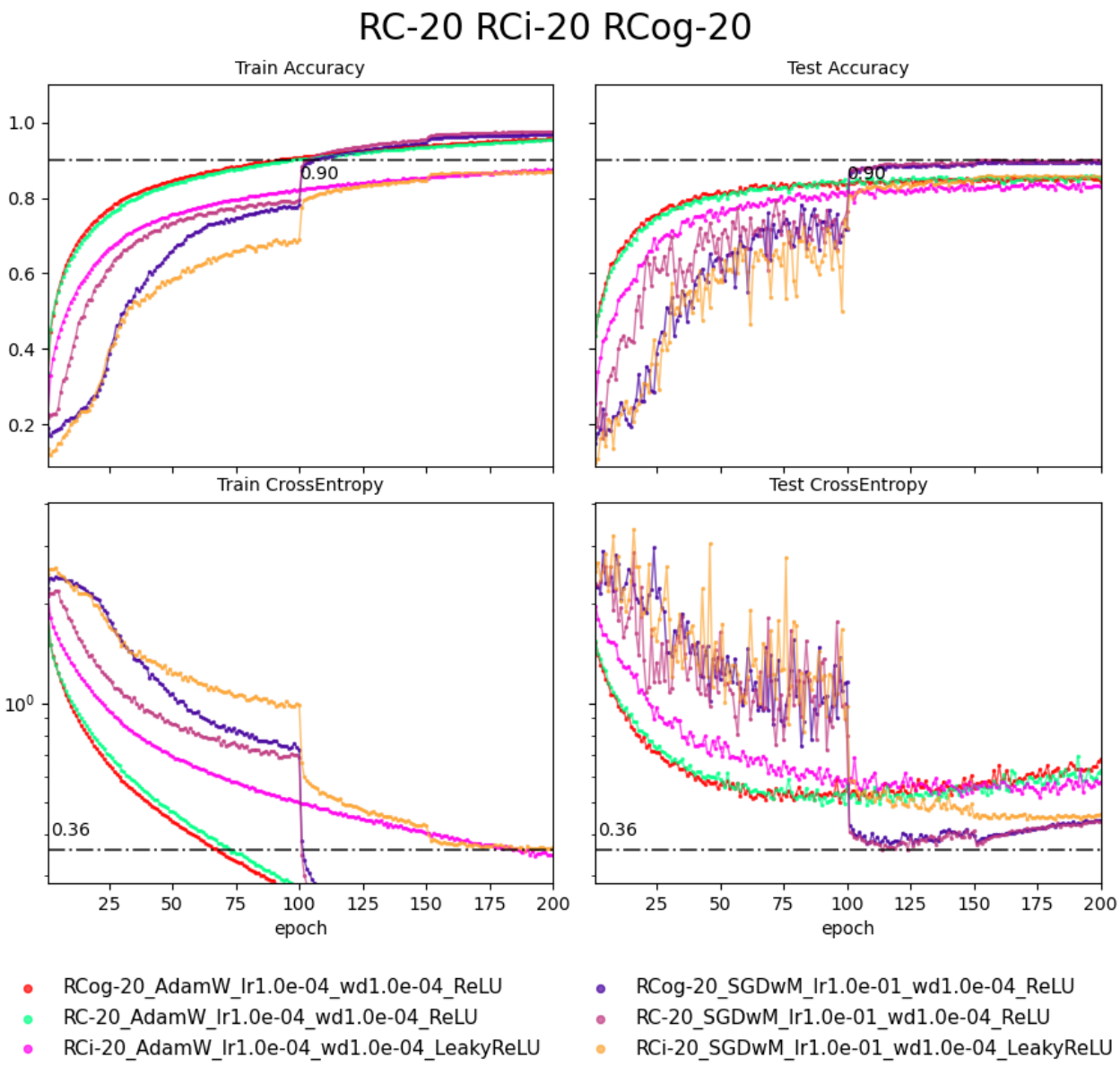

Figure 12: Results of the