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*. The code is adapted to function without issues on CPU as well, albeit quite slower in comparison.

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, as well as nn.Dropout(p=0.3) as the very first layer.

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).

Iteratively, we tuned individua 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.

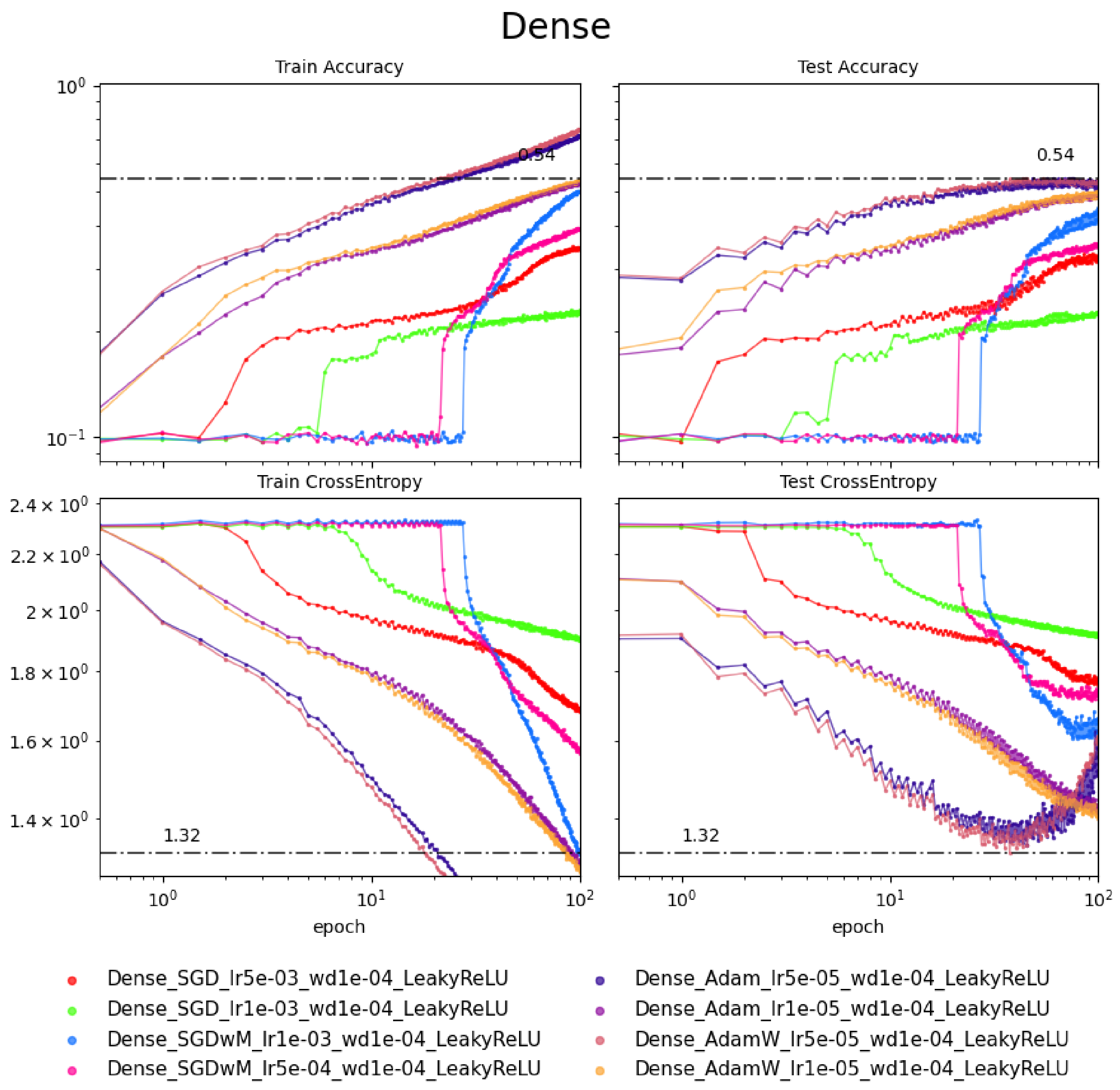

Figure 1: Plot of results of the preliminary exploration of FCNN architecture.

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.

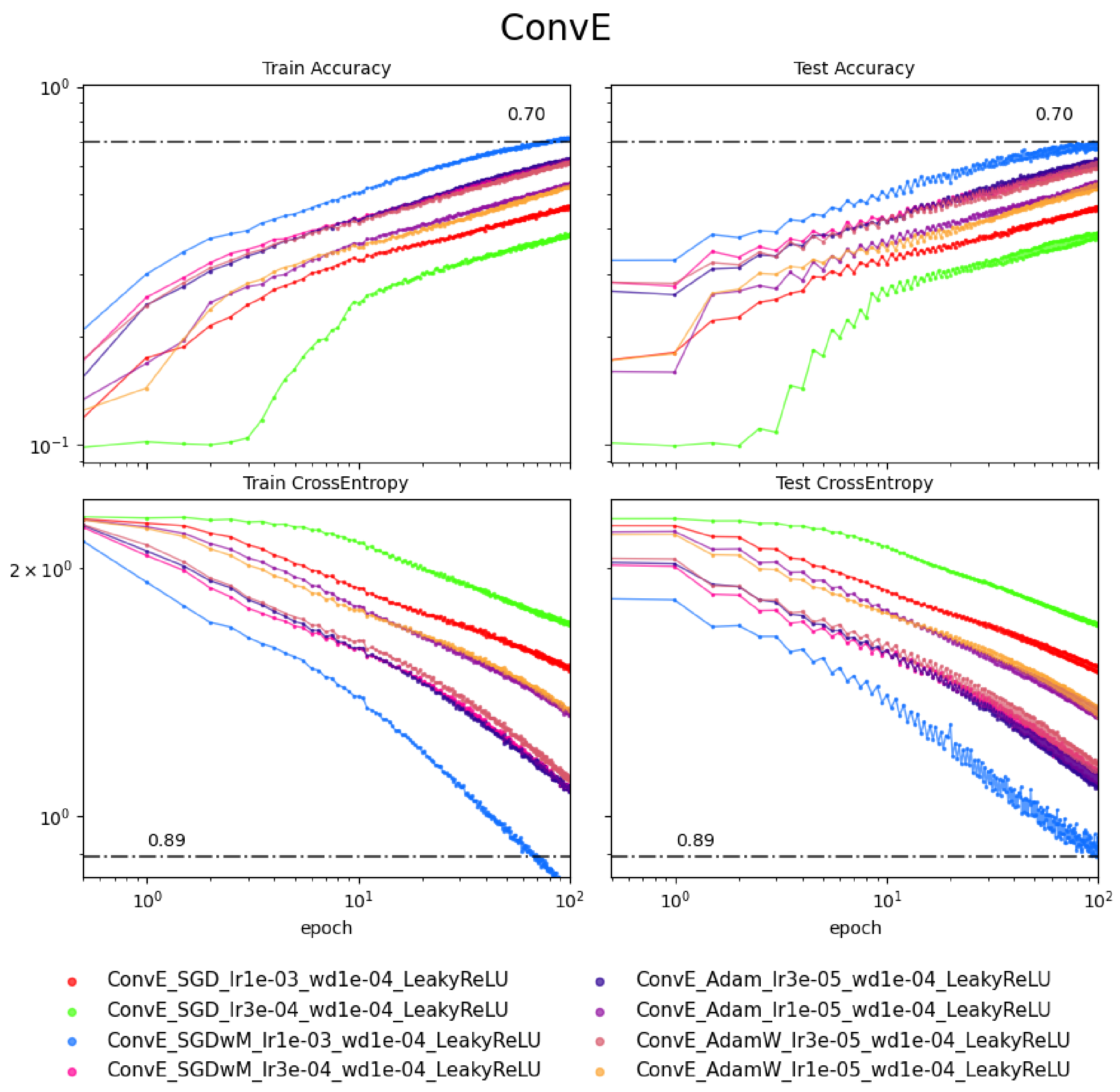

Figure 2: Plot of results of the preliminary exploration of ConvEncoder architecture.

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.

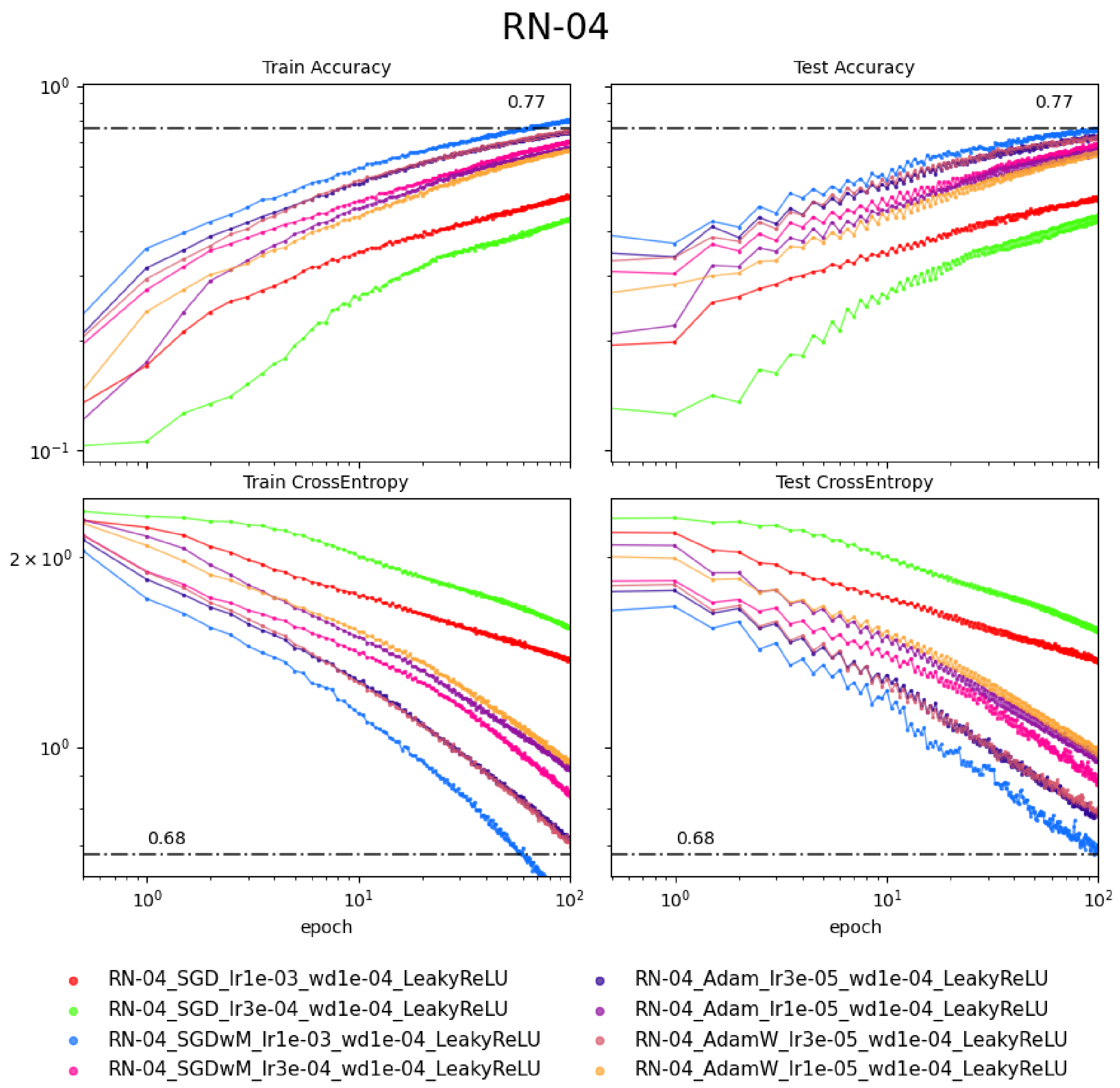

Figure 3: Plot of results of the preliminary exploration of RN04 with ConvEncoder architecture.

3 Depth and Learning Rate of RN variants exploration