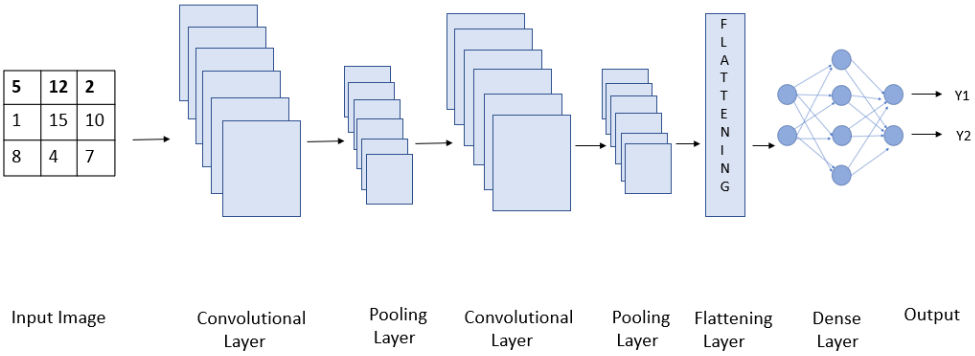
097209 ML2 – HW2 Report

Q1 :

- ***Model Architecture***: The model consists of three convolutional two fully connected layer. The convolutional layers have a kernel size of 5 for the first two layers and 3 for the third layer. The padding is set to 2 for all the CNN layers. The number of filters in the convolutional layers increases from 16 to 32 to 64. The model also includes batch normalization and ReLU activation layers after each convolutional layer. The model ends with 2 fully connected layers with 16 and 10 output units and a log SoftMax activation function.

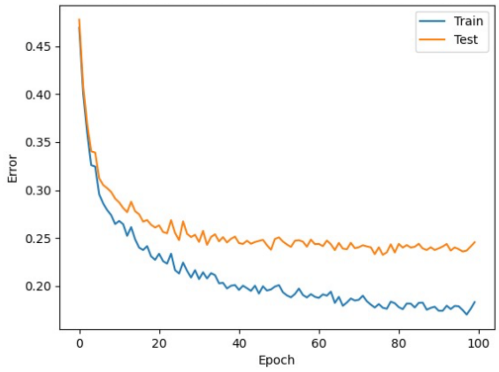


- ***Number of parameters***: 49,338 < 50,000
- ***Training Procedure***: The model is trained for a total of 90 epochs with a batch size of 128.

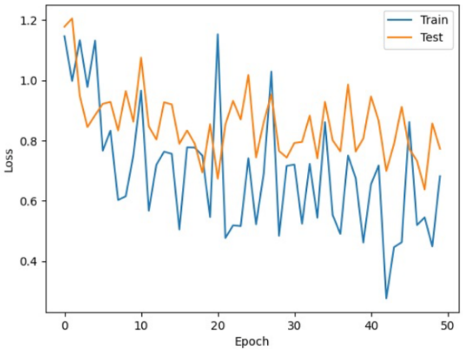
The CIFAR10 dataset consists of 60,000 32x32 color training images and 10,000 test images, labeled into 10 classes. It is a relatively small dataset, which makes it prone to overfitting, especially when training deep learning models with many parameters.
One way to reduce overfitting and improve the generalization of the model is to use data augmentation. This can be done by applying various transformations to the training images, such as random cropping, horizontal flipping, and color jittering. These transformations can help the model learn more robust and generalizable features, which can improve its performance on the test set.
In addition to data augmentation, it is also important to use techniques such as regularization and early stopping to further reduce the risk of overfitting. Using a smaller model architecture and reducing the number of training epochs can also help to improve the generalization of the model.
Overall, by using data augmentation and other regularization techniques, it is possible to achieve good results (more than 80% in our case) on the CIFAR10 dataset, even with a relatively small amount of labeled data.

I played with the batch size to get the best result and the number of epochs ; I keep track at each epoch for the loss so I can stop at the best epoch to have the maximal results.

Here the plots for the error and the loss. We can see that the loss is very volatile maybe because our learning rate is not very stable (*need to check this theory*). We are also constating the overfitting of the dataset (*like we saw in the previous homework*)

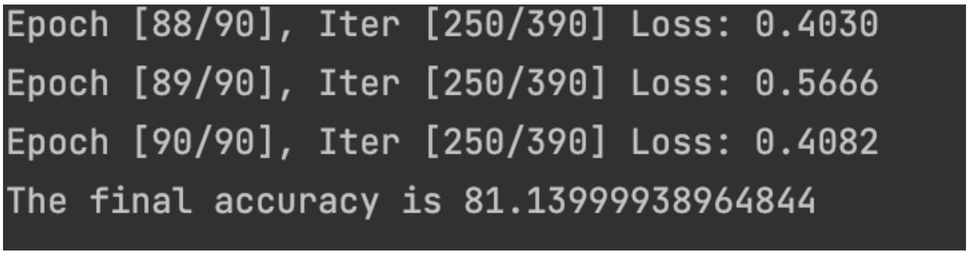


Error Per Epoch



Loss Per Epoch

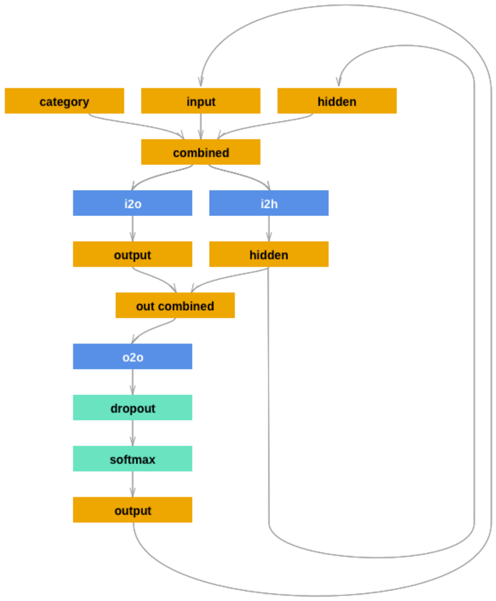
Final Results :



```
Epoch [88/90], Iter [250/390] Loss: 0.4030
Epoch [89/90], Iter [250/390] Loss: 0.5666
Epoch [90/90], Iter [250/390] Loss: 0.4082
The final accuracy is 81.13999938964844
```

Q2:

- ***RNN Model Architecture:*** The code defines an RNN class that extends the *nn.Module* class from PyTorch. It has several methods including init, forward, and *initHidden*. The init method initializes the class and defines several linear layers and a dropout layer. The forward method defines the forward pass of the RNN, which takes in a category tensor, an input tensor, and a hidden state tensor and returns an output tensor and a new hidden state tensor. The initHidden method returns a tensor of zeros with shape (1, hidden_size).



- ***Training Procedure***: Here a detailed list of the step for the code :

1. Read in the names data from files and build a dictionary of names per language.
2. Initialize the RNN model.
3. Train the RNN model using the train function.
4. Initialize the hidden state tensor to be a tensor of zeros.
5. Set the gradients of the parameters of the RNN model to zero.
6. Iterate through each element in the input tensor.
7. Run the RNN model with the category tensor, the current input element, and the hidden state tensor.
8. Calculate the loss between the output of the model and the target element.
9. Add the loss to the overall loss.
10. After iterating through all the input elements, calculate the gradients with respect to the loss.
11. Update the parameters of the model using the gradients and the learning rate.
12. Return the final output and the average loss per element.
13. Generate a name using the RNN model by calling the *generateName* function.
14. Pass in a category and a start letter to the *generateName* function.
15. Generate names for each language and start letter and print the results.

In this case, the goal of the RNN is to generate names based on nationality. To do this, the RNN takes in three inputs at each time step: the nationality category, the current letter, and the hidden state. The nationality category is a one-hot encoded vector that indicates the nationality of the name being generated. The current letter is the current input to the RNN at the current time step, and the hidden state is a summary of the previous inputs that the RNN has seen.

At each time step, the RNN produces two outputs: the next letter and the next hidden state. The next letter is the prediction that the RNN makes for the next letter in the name, based on the current input and the previous hidden state. The next hidden state is the updated summary of the previous inputs that the RNN will use as input at the next time step.

To train the RNN, we need to provide it with a set of input-output pairs for a given nationality. For each input-output pair, the input consists of the nationality category, a set of input letters, and a hidden state tensor, and the output consists of a set of target letters and a target hidden state tensor. The input letters and target letters are groups of consecutive letters from a name, such as ("A", "B") or ("B", "C"). The hidden state tensor and the target hidden state tensor are both tensors that summarize the previous inputs and are used as input at the next time step.

Overall, the RNN processes each input-output pair by using the input to predict the output at each time step, and it adjusts its internal parameters based on the prediction error between the predicted output and the target output. This process is repeated for each input-output pair in the training set, and the goal is to minimize the prediction error and produce accurate predictions for the next letter in the name.

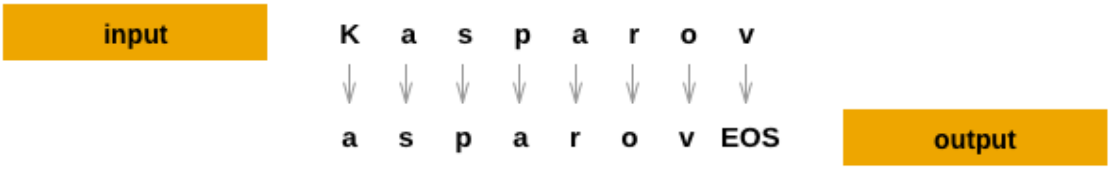
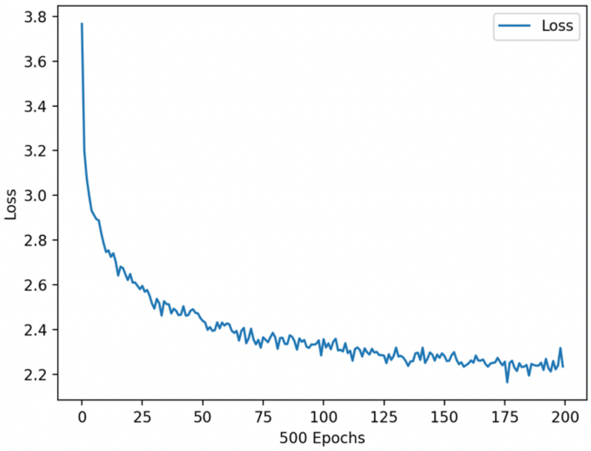
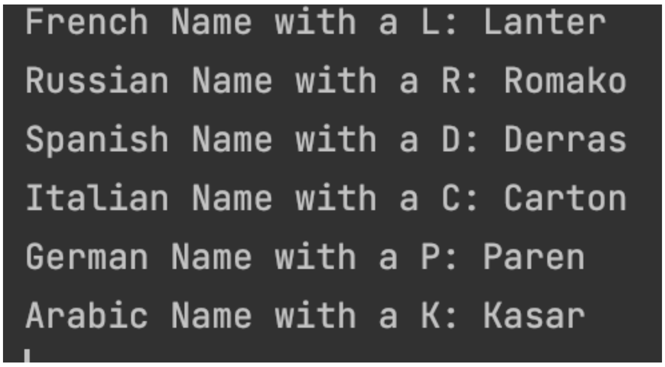


Image Taken from PyTorch Documentation



Loss Function per Epoch

Here the finals results for generating names :



```
French Name with a L: Lanter
Russian Name with a R: Romako
Spanish Name with a D: Derras
Italian Name with a C: Carton
German Name with a P: Paren
Arabic Name with a K: Kasar
```