Final Project

NLP: Surname Classification

PyTorch and Machine Learning

NCCU, Fall 2020

MBA | 107363015 | 郭育丞 (Morton Kuo)

2020 / 01/ 15

Contents

(1) Introduction & Pre-processing

1-1 **Code on Colab**

All the code is available on Colab.

1. Preprocessing: https://bit.ly/3hGWYhs
2. MLP: https://bit.ly/3nTLxpf
3. CNN: https://bit.ly/3rwRCKr
4. RNN: https://bit.ly/3ptfcG8

1-2 **Introduction to Data & Model**

Source: The data and models all come from Rao, D. & McMahan, B. (2019). Natural Language Processing with PyTroch. California, CA: O'Reilly Media.

Data: The surname dataset, a collection of 10,000 surnames from 18 different nationalities collected by the authors from different name sources on the Internet. We split the data into 70% training data, 15% validation data, and 15% test data.

Model: Build surname classifier through MLP, CNN and RNN.

[Remark] The models with given parameters in this book are taken as baseline models.

Let's check the architecture (or so-called anatomy) of PyTorch & Tensorflow.

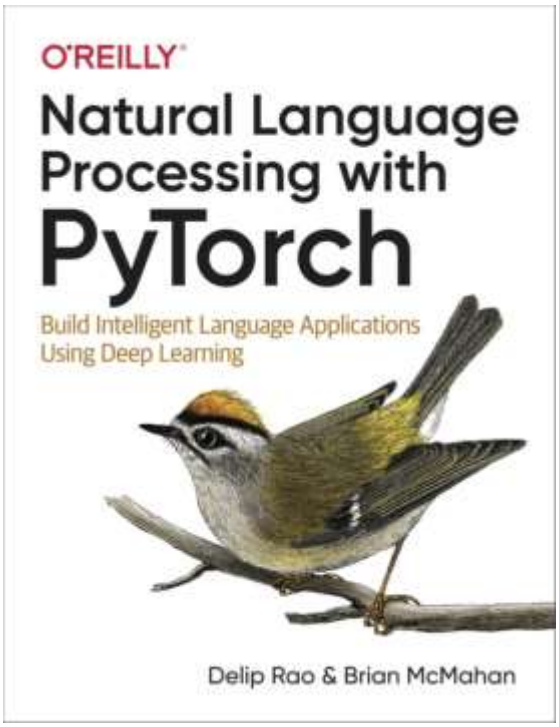

Figure 1: The Book "Natural Language Processing with PyTroch".

1-3 **Text Pre-processing Steps for NLP & Text Mining**

1. Expanding Contractions
2. Lower Case Transformation
3. Removing Non-alphabet Characters
4. Tokenization
5. Removing Stopwords
6. Lemmatization OR Stemming

Take a paragraph in "Alice in Wonderland" for instance:

The original paragraph => "I'm sure those are not the right words," said poor Alice, and her eyes filled with tears again as she went on, "I must be Mabel after all, and I shall have to go and live in that poky little house, and have next to no toys to play with, and oh! ever so many lessons to learn! No, I've made up my mind about it; if I'm Mabel, I'll stay down here!

The paragraph after preprocessing => ['sure', 'right', 'word', 'say', 'poor', 'alice', 'eye', 'fill', 'tear', 'go', 'must', 'mabel', 'shall', 'go', 'live', 'poky', 'little', 'house', 'next', 'toy', 'play', 'oh', 'ever', 'many', 'lesson', 'learn', 'make', 'mind', 'mabel', 'stay']

1-4 **Data Pre-processing for this Project**

The surname dataset is spitted into 70% training, 15% validation, 15% test data. Then, we vectorize the surnames using one-hot encoding. We utilize two variants—"collapsed one-hot vector" and "one-hot matrix".

1	surname	nationality
2	Woodford	English
3	Coté	French
4	Kore	English
5	Koury	Arabic
6	Lebzak	Russian
7	Obinata	Japanese
8	Rahal	Arabic
9	Zhuan	Chinese

Figure 2: The original data.

1	nationality	nationality_index	split	surname
2	Arabic	15	train	Totah
3	Arabic	15	train	Abboud
4	Arabic	15	train	Fakhoury
5	Arabic	15	train	Srour
6	Arabic	15	train	Sayegh
7	Arabic	15	train	Cham
8	Arabic	15	train	Haik

Figure 3: The data after some preprocessing procedures.

1-5 **One-hot Representation**

S_1: Time flies like an arrow.

S_2: Fruit flies like a banana.

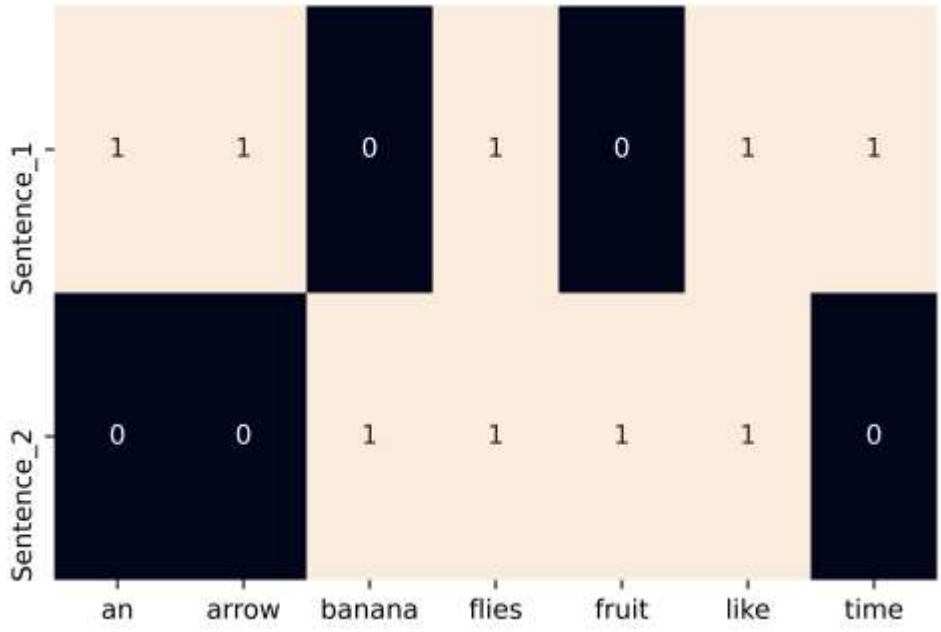

Figure 4: A One-hot encoding example.

1-6 **The Classification Models & Baseline Models**

We try to classify surnames from 18 different nationalities using NLP. Specifically, we leverage MLP, CNN, RNN (Elman RNN) to accomplish that. Then, we quickly walk through the basic concepts.

- Neuron: A minimum unit of neural network.

- Perceptron: A single-layer neural network.
- FNN (feedforward neural network): MLP (multilayer perceptron, also called "fully-connected" network), CNN (convolutional neural network).
- RNN (recurrent neural network): RNN, LSTM (long short-term memory), GRU (gated recurrent unit).
- Baseline Models: The MLP, CNN, RNN model retrieved from the book, only change the drop probability of RNN from 50% to 0%.

1-7 **The Tuning Deep Networks**

1. **Activators**: sigmoid(), tanh(), ReLU() [fast], PReLU(), LeakyReLU(), ELU(), SELU(), Softmax()

2. **Optimizers**: SGD, Momentum, AdaGrad, RMSprop, Adam, Nadam, AdaMax

3. **Batch Normalization (BN)**: Introduced by Ioffe & Szegedy in 2015, allowing models to be less sensitive to initialization of the parameters and simplifies the tuning of learning rates.

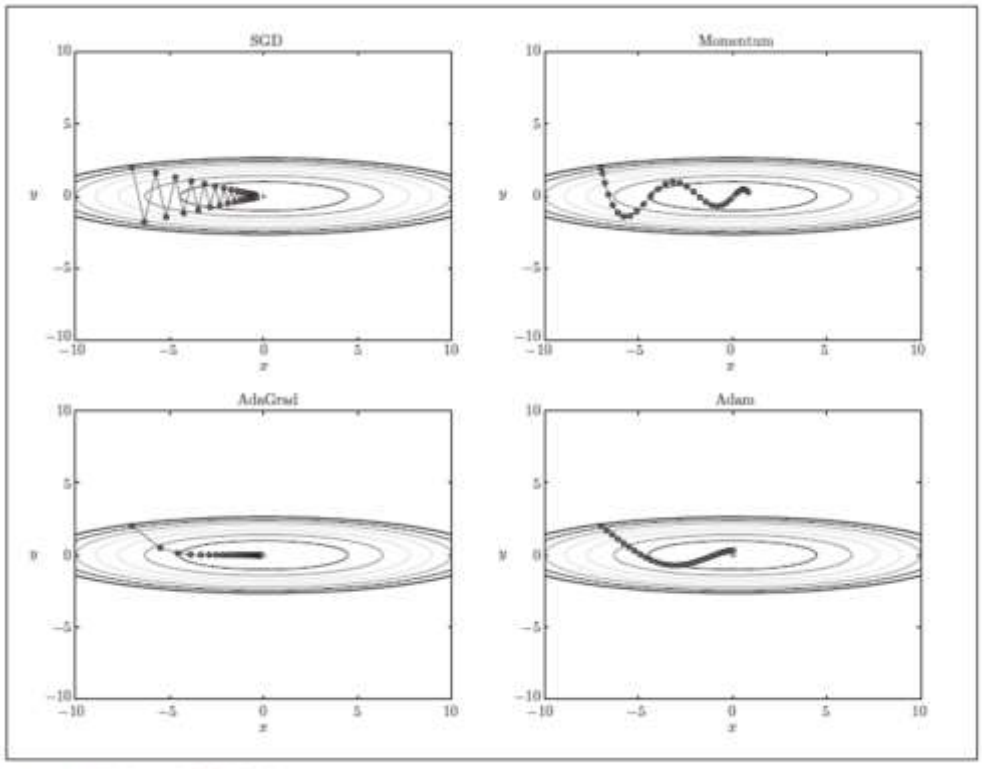

图6-8 最优化方法的比较：SGD、Momentum、AdaGrad、Adam

Figure 5: Comparison of different optimizers.

4. **Epochs and Mini-Batch Size**

- The relationship between how fast our algorithm can learn the model is typically U-shaped (batch size versus training speed). This means that initially as the batch size becomes larger, the training time will decrease. Eventually, we'll see the training time begin to increase when we exceed a certain batch size

that is too large.

- A larger mini-batch size means smoother gradients.
- For performance (this is most important in the case of GPUs), we should use a multiple of 32 for the batch size, or multiples of 16, 8, 4, or 2 if multiples of 32 can't be used (or result in too large a change given other requirements). In short, the reason for this is simple: memory access and hardware design are better optimized for operating on arrays with dimensions that are powers of two, compared to other sizes. For example, we should use a layer size of 128 over size 125, or size 256 over size 250, and so on.
- In practice, 32 to 256 is common for CPU training, and 32 to 1,024 is common for GPU training. Usually, something in this range is good enough for smaller networks, though you should probably test this for larger ones (where training time can be prohibitive).
- The Relationship Between Mini-Batch Size and Epochs: If we increase our mini-batch size by a factor of two, we need to increase the number of epochs by a factor of two in order to maintain the same number of parameter updates. The number of parameter updates per epoch is just the total number of examples in our training set divided by the mini-batch size.
- Using a larger mini-batch size might help our network to learn in some difficult cases, such as for noisy or imbalanced datasets.

5. **Dropout**

- Introduced by Hinton in 2012. Usually 10% ~ 50%. In CNN, often 40% ~ 50%. In RNN, usually 20% ~ 30%.
- Dropout & weight decay are all regularization techniques.
- The resulting neural network can be seen as an averaging ensemble of all these smaller neural networks.
- Sometimes we'll use no dropout at all on the input, especially for noisy or sparse datasets.
- It is not common to use dropout on the output layer.
- Dropout in all hidden layers works more effectively than in only one hidden layer.
- Dropout also tends to make the activations of hidden units sparse even when other regularization techniques are absent, leading to sparse representations.

6. **Dropout**

- L1 (Lasso) or L2(Ridge) regularization. In torch.optim, the parameter weight_decay (float, optional) adopts L2.
- Dropout & weight decay are all regularization techniques.
- L1: Sparse models. L1 has less of a penalty for large weights, but leads to many weights being driven to 0 (or very close to 0), meaning that the resultant weight vector can be sparse.
- L2: Dense models. L2 more heavily penalizes large weights, but doesn't drive small weights to 0.
- In practice, we see L2 regularization give better performance over L1 outside of explicit feature selection.

- We can combine L1 and L2 in the one network, i.e., Elastic Net (Zou et al., 2005).
- For cases in which we're using early stopping, we might not want to use L2 regularization at all, because early stopping is more efficient at performing the same mechanics as L2 (Bengio, 2012).

(2) MLP

At pre-processing stage of MLP, we utilize "collapsed one-hot vector", which do not count the frequency of characters and only record the character appearing or not. Also, this approach ignores the sequential information.

2-1 **MLP: Baseline Model**

◆ MLP Baseline Model structure

(0): Linear(input_dim, hidden_dim, bias=True)

(1): ReLU()

(2): Linear(hidden_dim, output_dim, bias=True)

◆ Parameters of MLP Baseline Model

Epoch = 25

Hidden layer dimension = 300 (only one hidden layer)

Batch Size = 64

Activator = ReLU()

Optimizer = Adam

Learning rate = 0.001 (commonly recommended)

Learning rate schedule: lr_scheduler.ReduceLROnPlateau()

No batch normalization, dropout, weight decay

◆ MLP Baseline

Train Accuracy: 52.73%

Valid Accuracy: 46.31%

Test Accuracy: 46.69%

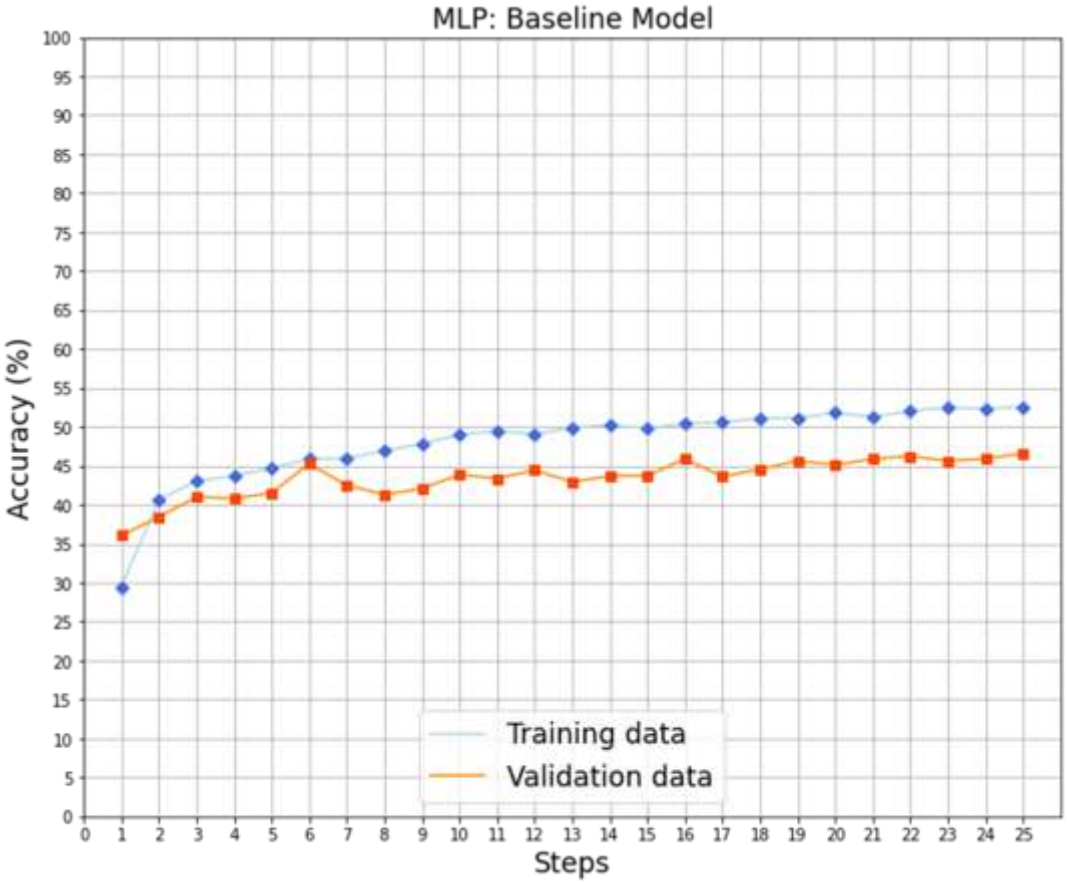

Figure 6: MLP baseline model.

2-2 **MLP: Improving Baseline Model**

What could we do to further improve the baseline model?

- **Nonlinear Activators**: Without nonlinearity between two Linear layers, two Linear layers in sequence are mathematically equivalent to a single Linear layer.
- **Batch Normalization (BN)**: Introduced by Ioffe & Szegedy in 2015, allowing models to be less sensitive to initialization of the parameters and simplifies the tuning of learning rates.
- **Dropout**: Introduced by Hinton in 2012. Usually 10% ~ 50%. In CNN, often 40% ~ 50%. In RNN, usually 20% ~ 30%. It's just like operating ensemble learning by a NN.
- **Weight Decay**: Use L1 (Lasso) or L2(Ridge) regularization. In general, L2 is recommended. In torch.optim, the parameter weight_decay (float, optional) adopts L2.

2-3 **MLP: Best Model**

- FNN/MLP Best Model structure

(0): Linear(input_dim, hidden_dim, bias=True)

(1): LeakyReLU()

(2): Linear(hidden_dim, output_dim, bias=True)

- Parameters of FNN/MLP Best Model

Epoch = 25

Hidden layer dimension = 1000 (only one hidden layer)

Batch Size = 2

Activator = LeakyReLU()

Optimizer = Adam

Learning rate = 0.001 (commonly recommended)

Learning rate schedule: lr_scheduler.ReduceLROnPlateau()

No batch normalization

Dropout = 0.0

Weight decay = 0.0

- MLP Best

Train Accuracy: 93.63%

Valid Accuracy: 67.26%

Test Accuracy: 66.87%

We tried batch norm, dropout, weight decay and adding a hidden layer, but oddly none of these offer us better performance. We found LeakyReLU() performs better than PReLU() and ReLU().

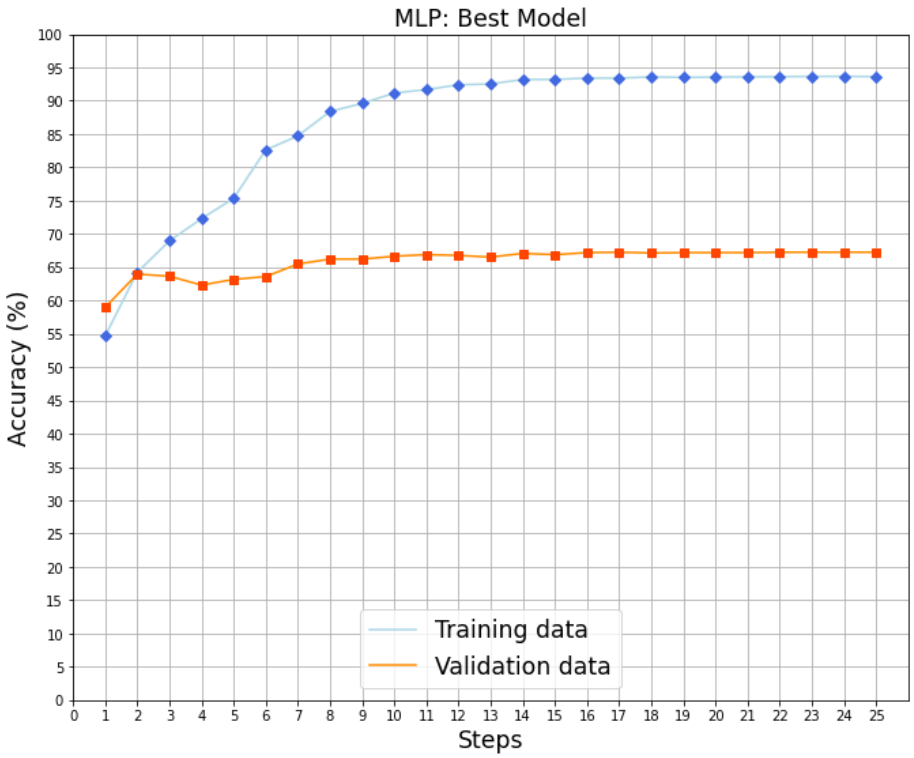

Figure 7: MLP best model.

2-4 **MLP: Summary**

- MLP Baseline

Train Accuracy: 52.73%

Valid Accuracy: 46.31%

Test Accuracy: 46.69%

- MLP Best

Train Accuracy: 93.63%

Valid Accuracy: 67.26%

Test Accuracy: 66.87%

✓ Regularization approaches (dropout & weight decay using L2) don't help at all! They are all 0 in the best MLP model.

✓ Probably since the surname dataset is not big enough to generate overfitting!

(3) CNN

At pre-processing stage of CNN, we utilize “one-hot matrix”, which retains the sequential information.

3-1 **CNN: Baseline Model**

◆ CNN Baseline Model structure

(0): Conv1d(in_channels, num_channels, bias=True, kernel_size=3)

(1): ELU()

(2): Conv1d(num_channels, num_channels, bias=True, kernel_size=3, stride=2)

(3): ELU()

(4): Conv1d(num_channels, num_channels, bias=True, kernel_size=3, stride=2)

(5): ELU()

(6): Conv1d(num_channels, num_channels, bias=True, kernel_size=3)

(7): ELU()

(8): Linear(num_channels, num_classes)

◆ Parameters of CNN Baseline Model

Epoch = 30

Number of channel = 256

Batch Size = 128

Activator = ELU()

Optimizer = Adam

Learning rate = 0.001 (commonly recommended)

Learning rate schedule: lr_scheduler.ReduceLROnPlateau()

No dropout, weight decay

◆ CNN: Baseline

Train Accuracy: 68.83%

Valid Accuracy: 57.16%

Test Accuracy: 56.38%

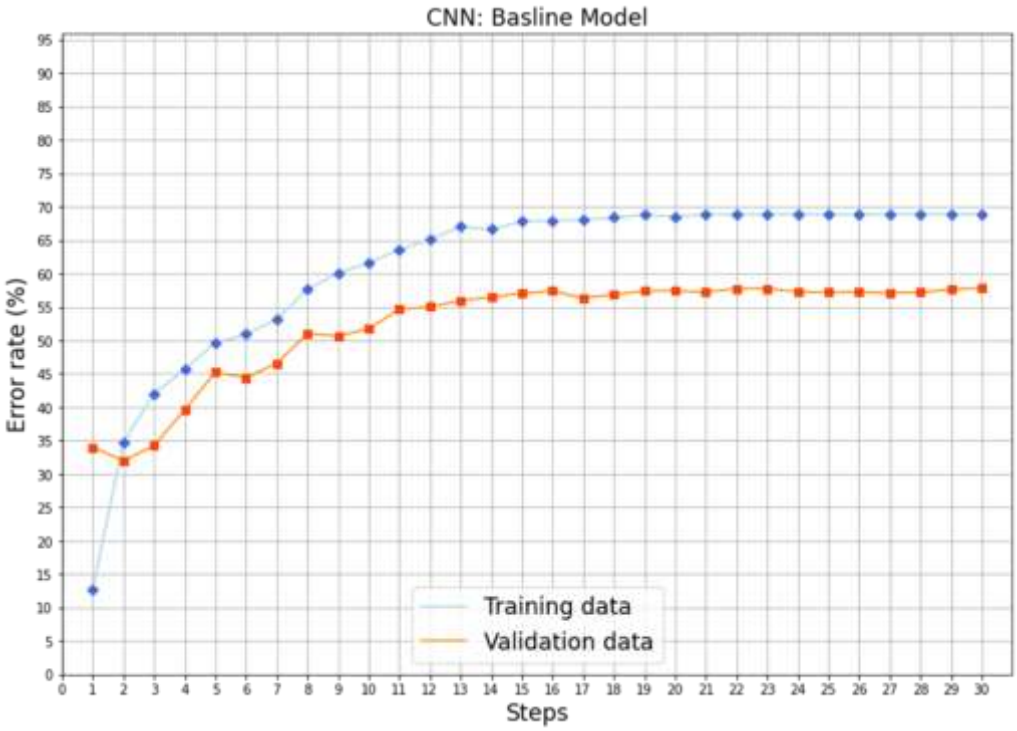

Figure 8: CNN baseline model.

3-2 **CNN: Best Model**

◆ CNN Best Model structure

(0): Conv1d(in_channels, num_channels, bias=True, kernel_size=3)

(1): BatchNorm1d(num_channels)

(3): ELU()

(4): Conv1d(num_channels, num_channels, bias=True, kernel_size=3, stride=2)

(5): BatchNorm1d(num_channels)

(6): ELU()

(7): Conv1d(num_channels, num_channels, bias=True, kernel_size=3, stride=2)

(8): BatchNorm1d(num_channels)

(9): ELU()

(10): Conv1d(num_channels, num_channels, bias=True, kernel_size=3)

(11): BatchNorm1d(num_channels)

(12): ELU()

(13): Linear(num_channels, num_classes)

◆ Parameters of CNN Baseline Model

Epoch = 30

Number of channel = 280

Batch Size = 8

Activator = ELU()

Optimizer = Adam

Learning rate = 0.001 (commonly recommended)

Learning rate schedule: lr_scheduler.ReduceLROnPlateau()

Weight decay = 5e-5 (L2 regularization)

Dropout = 0.0

◆ CNN Best

Train Accuracy: 90.66%

Valid Accuracy: 76.46%

Test Accuracy: 71.20%

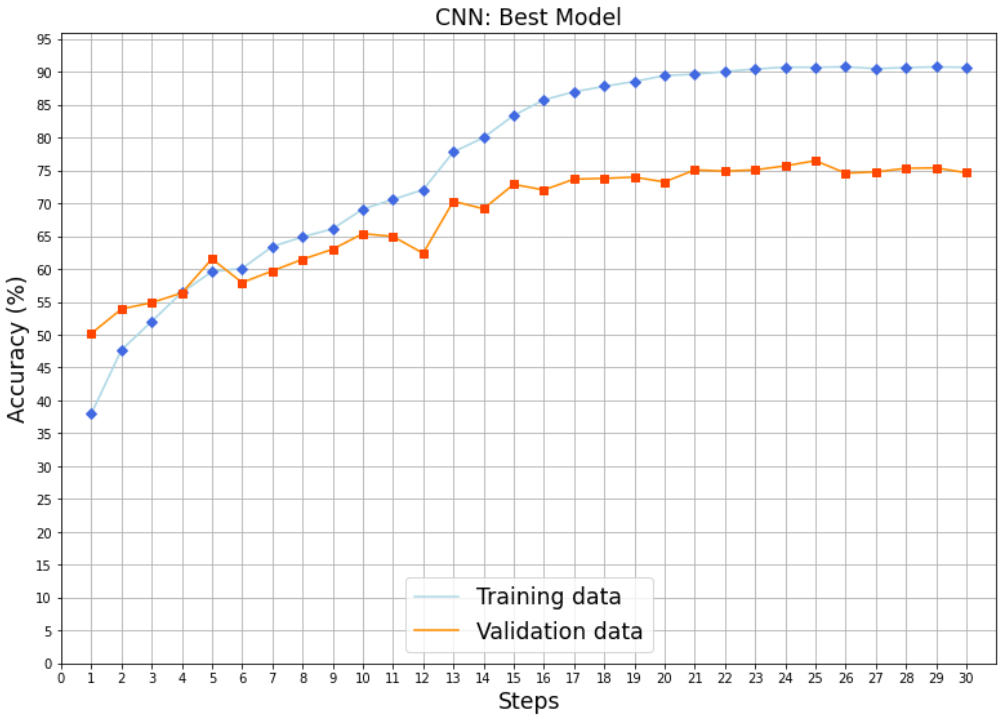

Figure 9: CNN best model.

3-3 **CNN: Summary**

◆ CNN: Baseline

Train Accuracy: 68.83%

Valid Accuracy: 57.16%

Test Accuracy: 56.38%

◆ CNN Best

Train Accuracy: 90.66%

Valid Accuracy: 76.46%

Test Accuracy: 71.20%

✓ Regularization approaches (dropout & weight decay using L2) barely help! In the best CNN model,

1. Drop probability = 0
2. Weight decay = 5e-5

✓ Probably since the surname dataset is not big enough to generate overfitting!

(4) RNN

At pre-processing stage of RNN, we utilize "one-hot matrix", which retains the sequential information.

4-1 **RNN: Baseline Model**

◆ RNN Baseline Model structure

(1): nn.Embedding(num_embeddings=num_embeddings,embedding_dim=embedding_size, padding_idx=padding_idx)

(2): ElmanRNN(input_size=embedding_size, hidden_size=rnn_hidden_size, batch_first=batch_first)

(4): nn.Linear(in_features=rnn_hidden_size,out_features=rnn_hidden_size)

(5): nn.ReLU()

(6): nn.Linear(in_features=rnn_hidden_size, out_features=num_classes)

◆ Parameters of RNN Baseline Model

Epoch = 30

char_embedding_size= 100

rnn_hidden_size= 64

Batch Size = 64

Activator = ReLU()

Optimizer = Adam

Learning rate = 1e-3 (commonly recommended)

Learning rate schedule: lr_scheduler.ReduceLROnPlateau()

Weight decay = 0 (L2 regularization)

Dropout = 0.0 (Originally 0.5 but 0.5 results in worse outcome, so I make it 0.)

No batch normalization

◆ RNN Baseline

Train Accuracy: 65.25%

Valid Accuracy: 56.81%

Test Accuracy: 56.38%

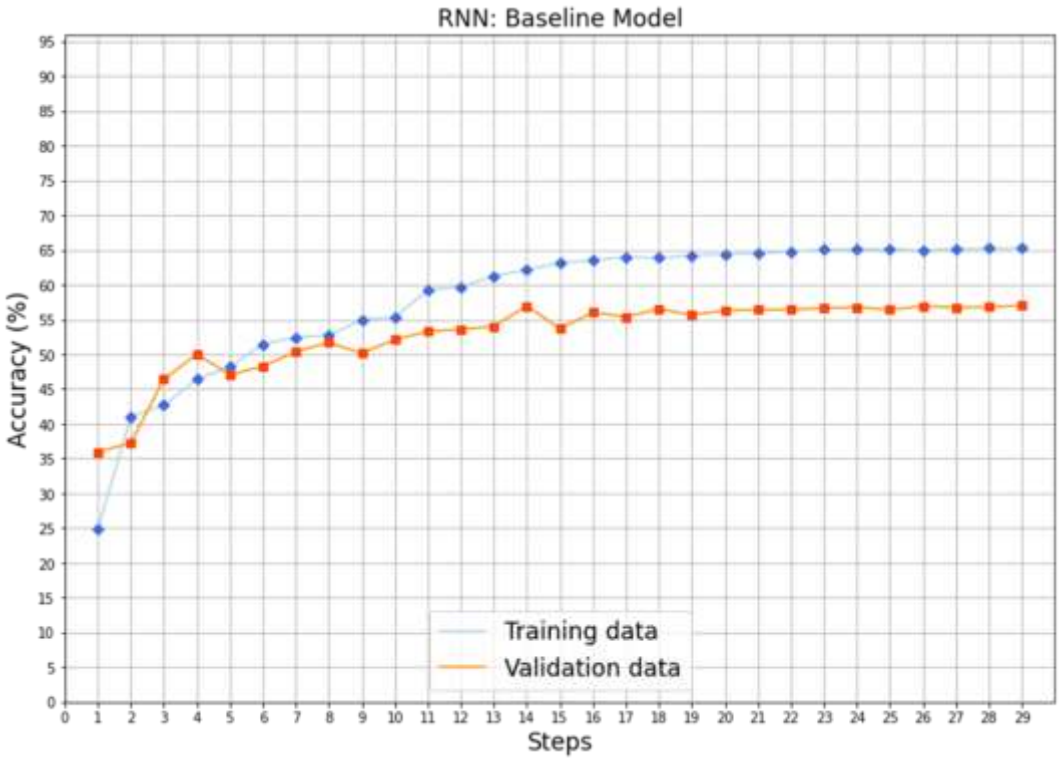

Figure 10: RNN baseline model.

4-2 **RNN: Best Model**

◆ RNN Baseline Model structure

(1): nn.Embedding(num_embeddings=num_embeddings embedding_dim=embedding_size, padding_idx=padding_idx)

(2): ElmanRNN(input_size=embedding_size, hidden_size=rnn_hidden_size, batch_first=batch_first)

(4): nn.Linear(in_features=rnn_hidden_size,out_features=rnn_hidden_size)

(5): nn.BatchNorm1d(rnn_hidden_size)

(6): nn.LeakyReLU()

(7): nn.Linear(in_features=rnn_hidden_size, out_features=num_classes)

◆ Parameters of RNN Baseline Model

Epoch = 30

char_embedding_size= 300

rnn_hidden_size= 900

Batch Size = 32

Activator = LeakyReLU()

Optimizer = Adam

Learning rate = 1e-3 (commonly recommended)

Learning rate schedule: lr_scheduler.ReduceLROnPlateau()

Weight decay = 1e-4 (L2 regularization)

Dropout = 0.0

◆ RNN Best

Train Accuracy: 94.23%

Valid Accuracy: 72.37%

Test Accuracy: 63.91%

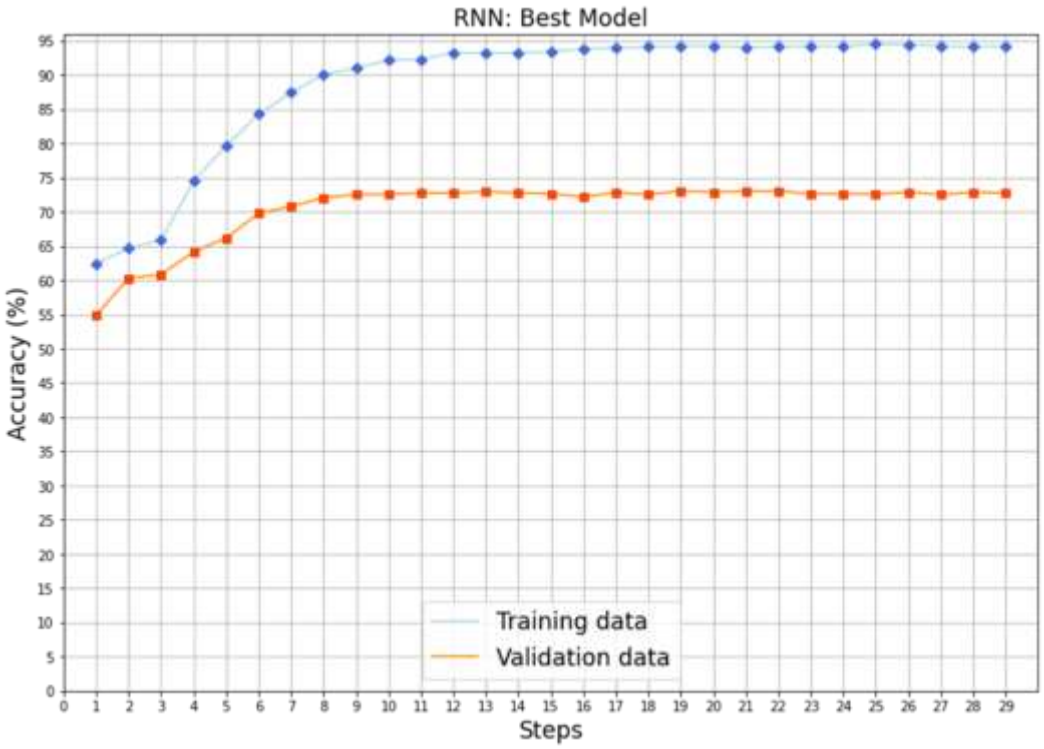

Figure 11: RNN best model.

4-3 **RNN: Summary**

◆ RNN Baseline

Train Accuracy: 65.25%

Valid Accuracy: 56.81%

Test Accuracy: 56.38%

◆ RNN Best

Train Accuracy: 94.23%

Valid Accuracy: 72.37%

Test Accuracy: 63.91%

✓ Regularization approaches (dropout & weight decay using L2) barely help! In the best RNN model,

1. Drop probability = 0
2. Weight decay = 1e-4

✓ Probably since the surname dataset is not big enough to generate overfitting!

(5) Conclusion

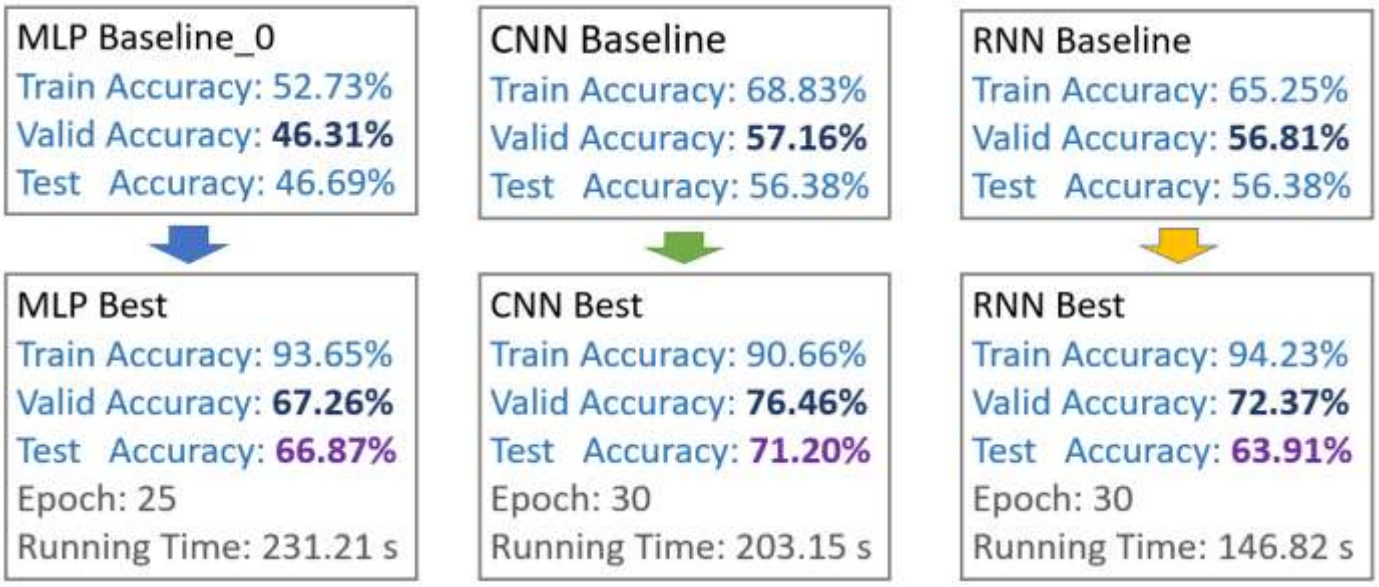

Figure 12: Comparison of all models

- 1. **MLP vs. CNN**

Compared to a fully connected layer, a convolution layer in a CNN has a much smaller number of parameters. This allows us to build deeper models without worrying about memory overflow. Also, deeper models usually lead to better performance.

- 2. **CNN vs. RNN**

One-dimensional convolutions ***sometimes perform better*** than RNNs and are ***computationally cheaper***.

- 3. **MLP, CNN, RNN — Ensemble Learning**

"MLP Best", "CNN Best" and "RNN best" seem to fit distinct patterns of "surname classification" respectively, so ensemble learning would help generating a more performant NN model.

- 4. **Prospect**

To achieve better test accuracy, we may do as follows:

1. MLP: Making a deeper MLP.

2. CNN: Choose more complex CNN architecture, like ResNet.

3. RNN: The RNN we utilize is a vanilla RNN: Elman RNN. LSTM and GRU would probably perform better than Elman RNN.

4. Optimizer: We used Adam here, and we could try others.

5. Activator: We used ReLU(), LeakyReLU(), ELU(). We may try others.

6. Regularization: We found the surname dataset is not huge enough to yield overfitting, so no need to spend time on this.

(6) Reference

1. Géron, A. (2019). Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd ed.). California, CA: O'Reilly Media.

2. Rao, D. & McMahan, B. (2019). Natural Language Processing with PyTroch. California, CA: O'Reilly Media.

3. Sarkar, D. (2019). Text Analytics with Python (2nd ed.). Karnataka, India: Apress.

4. Ganegedara, T. (2018). Natural Language Processing with TensorFlow. Birmingham, UK: Packt Publishing.

5. Subramanian, V. (2018). Deep Learning with PyTorch. Birmingham, UK: Packt Publishing.0

6. Patterson, J. & Gibson, A. (2017). Deep Learning: A Practitioner's Approach. California, CA: O'Reilly Media.

7. 斎藤康毅 (2016). ゼロから作る Deep Learning ―Python で学ぶディープラーニングの理論と実. Japan, JP: O'Reilly Japan.

8. Kuo, M. (2021). ML16: Hands-on Text Preprocessing. Retrieved from

https://medium.com/analytics-vidhya/ml16-ef6105b5bb34#6ef4