TEXAS A&M UNIVERSITY
Engineering

An introduction on building deep models

CSCE-638 Natural Language Processing

Yaochen Xie

An Overview of PA2

- Goal: implement the following methods to classify the sentiment of movie review texts
  - Naive Bayes
  - Perceptron
  - Hierarchical Attention Network (optional)

An Overview of PA2

TEXAS A&M UNIVERSITY Engineering

- Pipeline of implementation
  - Data (pre-)processing: write functions to convert documents into features and labels (+/-). Features for Naive Bayes and Perceptron are based on word frequency/presence.
  - Learning: estimate or optimize parameters given the processed data, in *model.train()*.
  - Inference/prediction: compute predicted class given a new data sample, in *model.classify()*.

An Overview of PA2

- Hierarchical Attention Network (optional)
  - Each document is initially represented as a tensor/array of shape *[1, num_sentence (20), num_words (50)]*. Inputs:
    1. Word tokens: each word is represented by an integer, i.e., the index of the word in the vocabulary.
    2. GloVe embedding: each word is represented by a vector.
    3. BERT/ELMO: each word is represented by a vector from pre-trained deep language models.

Hierarchical Attention Network

- Word token: given a vocabulary (dictionary) of size K, each word is an integer from [0, K-1].
- To input to the GRU/attention operator, we need all words to be represented by vectors → the embedding layer:
  - Maps word tokens into word embeddings. E.g., *torch.nn.Embedding(...)*, *tf.keras.layers.Embedding(...)*, *tf.nn.embedding_lookup(...)*.

- BERT and ELMO implementation:
  - Hugging-face (https://github.com/huggingface/transformers)
  - Works with both PyTorch and TensorFlow

```
>>> from transformers import AutoTokenizer, AutoModel

>>> tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
>>> model = AutoModel.from_pretrained("bert-base-uncased")

>>> inputs = tokenizer("Hello world!", return_tensors="pt")
>>> outputs = model(**inputs)
```

Hierarchical Attention Network

- Recap on the attention operation:
  - Intuition: given a query, aggregate information among candidates by weighted sum based on similarity/relevance between each candidate and the query.
  - Query: *q_1, …, q_n*; Key: *k_1, …, k_m*; Value: *v_1, …, v_m*

$$o_j = \frac{1}{C}\sum_{i=1}^{m} f(q_j, k_i) g(v_i)$$

  - HAN: query - a learned vector, value and key - from word embedding.

Hierarchical Attention Network

- Attention operations:
  - Self-attention v.s. attention with learned query
  - Additional resources:
    - http://people.tamu.edu/~sji/classes/attn-slides.pdf
    - http://people.tamu.edu/~sji/classes/attn.pdf

Hierarchical Attention Network

- An example of the attention implementation (pseudo-code):

```
x = concat([GRU_left(x), GRU_right(x)], dim=-1) # [B, S, W, D]
query = Parameter(shape=[1, D], trainable=True) # model parameters
key = x.view(B*S, W, D); value = Linear(x.view(B*S, W, D))
att_score = batch_matrix_mul(query, key.tanspose([0,2,1])) #[B*S, 1, W]
att_score = Softmax(att_score, dim=1) # or any other normalization
out = batch_matrix_mul(att_score, value) #[B*S, 1, D]
out = out.view(B, S, D) # summary vector of a document
```

- Check details in the paper!

TEXAS A&M UNIVERSITY
Engineering

Building Deep Learning Environments

Spinning a GPU for you Deep Models

- Linux GPU server if applicable
- Google Colab: https://colab.research.google.com/
  - *Jupyter-like platform*
  - *Pros: Free GPUs, easy setup ; Con: limited use time*
- AWS/Azure DL instances
  - *Similar to GPU server, but not free after trial*
- Your Windows laptop or desktop equipped with Nvidia GPU

Manage Environment

- Virtual environment for specific projects
  - Anaconda (recommended, https://www.anaconda.com/)
  - venv (https://docs.python.org/3/library/venv.html)
- Detached training: tmux, screen, *etc.*
- Installing DL frameworks
  - Tensorflow: https://www.tensorflow.org/install
  - PyTorch: https://pytorch.org/get-started/locally/

TEXAS A&M UNIVERSITY Engineering

Building Your Neural Network: An example with PyTorch

Pipeline of Building a Deep Model

- Things you need in your code (https://pytorch.org/tutorials/beginner/basics/quickstart_tutorial.html)
  - Dataset and dataloader
  - Your neural network model (class)
  - Training objective (loss function)
  - Optimizer, optimizing strategies

Dataset and Dataloader

- Dataset: a list of prepared data samples, or a indexable instance
  - E.g., *[(features_1, label_1), …, (features_N, label_N)]*
  - "*class CustomDataset(Dataset)*"
- DataLoader: generates a batch of data at each iteration
  - *train_dataloader = DataLoader(training_data, batch_size=64, shuffle=True)*
  - *for batch in train_dataloader:*

    (perform training with batch)

https://pytorch.org/tutorials/beginner/basics/data_tutorial.html

Your neural network model

- Write classes based on parent class "*torch.nn.Module*" and init parent class at the very beginning,
- Define operators and parameters in "*__init__*", use "*torch.nn.ModuleList*" instead regular list,
- Write operations with defined operators in "*forward*"
- https://pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html

```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10)
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits

model = NeuralNetwork().to(device)
print(model)
```

Training

- Select hyper-parameters: learning rate, batch size, epochs, *etc*.
- Define dataloader, model, loss function, and optimizer
- Set model to training mode and move to GPU
- https://pytorch.org/tutorials/beginner/basics/optimization_tutorial.html

model.train(); model.to(device); X.to(device); y.to(device)

```python
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

TEXAS A&M UNIVERSITY
Engineering

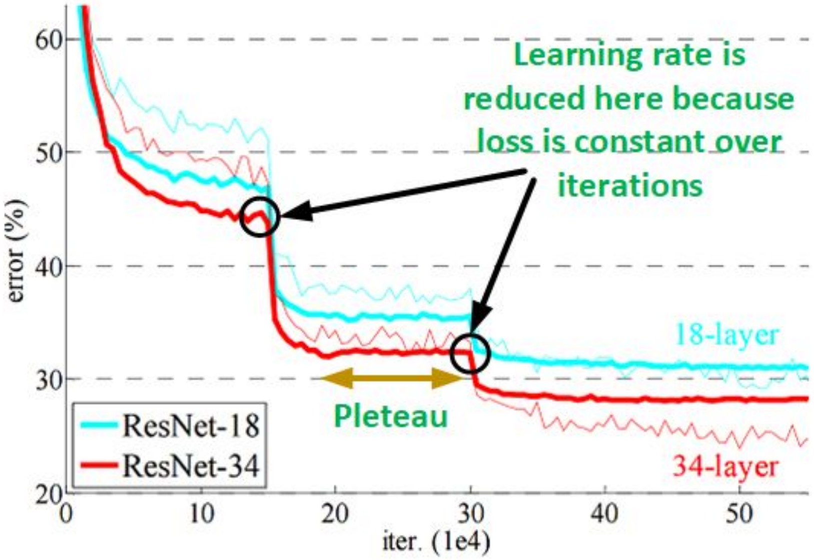
Some Training Tricks

Optimizer Selection

- Two most common optimizer: SGD v.s. Adam
  - Stochastic gradient descent (SGD):
    - Simple yet effective, the most stable optimizer
    - Works in most case
  - Adam:
    - Use adaptive learning rate for different parameters
    - Converge faster but is unstable sometime
- https://ruder.io/optimizing-gradient-descent/

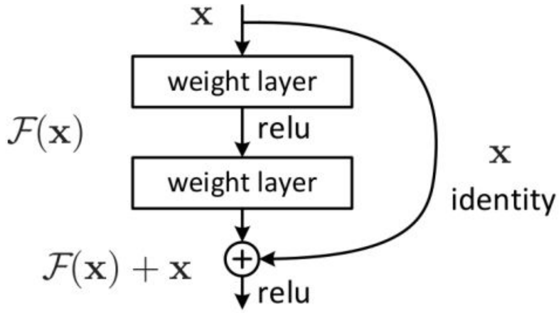
Learning Rate Scheduler

- Reduce learning rate after a certain number of epochs, or change learning rate over epochs.
- Commonly used scheduler:
  - ReduceOnPlateau, Step, Exponential, Cosine
- https://towardsdatascience.com/learning-rate-schedules-and-adaptive-learning-rate-methods-for-deep-learning-2c8f433990d1
- Also search for learning rate warmup.

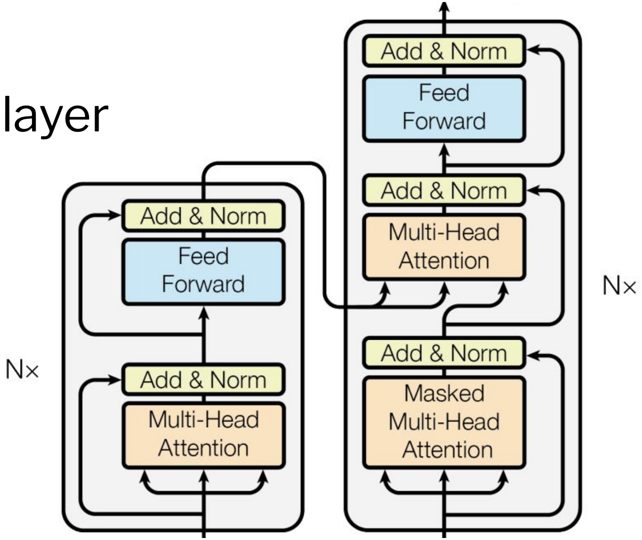


Tricks for Very Deep Models

- Skip/residual connection
  - Combines the input and output of a block
  - Allow the FP to skip certain blocks
  - Allow gradient to propagate to every layer



He et al. Deep Residual Learning for Image Recognition. CVPR 2016.



Vaswani et al. Attention Is All You Need. NIPS 2017.

Tricks for Very Deep Models

- Batch Normalization (Layer Normalization)
  - Makes training more stable, avoid over-smoothing issues, reduce overfitting
  - https://arxiv.org/abs/1502.03167

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

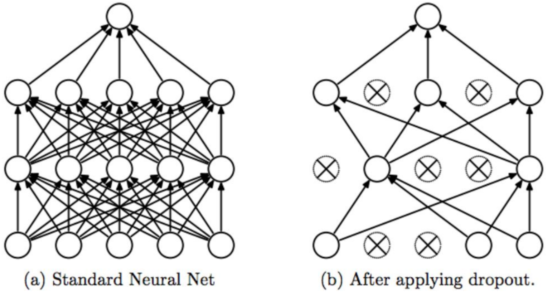
$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

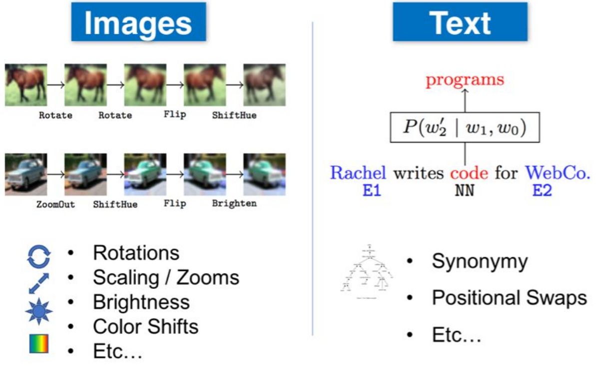
$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Tricks to Reduce Overfitting

- Try overfitting the model on training set (to evaluate the expressiveness); then address the overfitting issue.
- Regularizations
  - L1/L2 regularization (weight decay): add penalty to large values in parameter
  - Dropout: disable certain output dimensions during training
- Augmentation, early stopping, etc.



https://www.tech-quantum.com/implementing-drop-out-regularization-in-neural-networks/



https://madewithml.com/courses/mlops/augmentation/

TEXAS A&M UNIVERSITY
Engineering

Suggestions on Proposing & Building Deep Models

1. Model/method design first, hyper-tuning later

- Your model design should be based on empirical insights, reasonable motivations, theoretical frameworks, etc.
- If the method itself works, you don't even need much efforts on hyper-tuning.

2. Start from the simplest model when your methods consist of multiple (novel) components

- Always choose a simple base model to start with and consider it as the baseline.
- Add one component at a time to evaluate its effectiveness.

3. Reasoning the cause before making modifications when the model does not work as expected

- Observations → hypothesis → verification
  - Obtain enough evidence that gives you an idea on model behavior. E.g., visualizations, intermediate variables.
  - Make reasonable hypotheses on why models don't work.
  - Modify certain parts to verify the hypothesis.

TEXAS A&M UNIVERSITY

Engineering

Click to edit title style

Click to edit title style