MODERN INTELLIGENCE LEARNING ALGORITHMS

A PREPRINT

Tong Jia*
Artificial Intelligence Research
Beijing, China
cecilio.jia@gmail.com

March 1, 2020

ABSTRACT

Artificial intelligence technology has not only achieved great success in many theoretical fields such as machine learning, deep learning, ensemble learning and reinforcement learning, but also in many application scenarioes such as computer vision, natural language processing, speech recognition and recommender system. In this article, we first formulate the theory framework of learning algorithms, then we describe and derivate some kinds of algorithms, and finally we introduce the usage of related software frameworks (e.g., TensorFlow[1], PyTorch[2], Keras[3], MXNet[4], Theano[5], Scikit-learn[6]) and numerical optimization solvers (e.g., Gurobi[7], Mosek[8], CPLEX[9]), also provide code implementations of specific models.

Keywords Artificial Intelligence · Learning Algorithm · Optimization

*GITHUB.COM/DESPERADO-JIA.

Contents

I Learning Algorithm Framework **15**

1 Learning Algorithm Framework **16**

1.1 Theoretical Learning Framework . . . 16

1.2 System Engineering Learning Framework . . . 16

2 Dataset **17**

2.1 Explanatory Variable . . . 17

2.1.1 Categorical Variable . . . 17

2.1.2 Numerical Variable . . . 17

2.2 Response Variable . . . 17

3 Objective Function **18**

3.1 Loss Function for Regression . . . 18

3.1.1 Squared Loss . . . 18

3.1.2 Absolute Loss . . . 18

3.1.3 Huber Loss . . . 18

3.2 Loss Function for Classification . . . 20

3.2.1 Hinge Loss . . . 20

3.2.2 Log-Likelihood . . . 20

3.2.3 Kullback-Leibler Divergence . . . 21

3.2.4 Cross Entropy . . . 21

3.2.5 Focal Loss . . . 22

3.2.6 Triplet Loss . . . 23

4 Optimization Algorithm **26**

4.1 First-order Gradient Optimization Methods . . . 26

4.1.1 SGD . . . 26

4.1.2 Momentum . . . 27

4.1.3 Nesterov Momentum . . . 28

4.1.4 AdaGrad . . . 29

4.1.5 RMSprop . . . 30

4.1.6 AdaDelta . . . 31

4.1.7 Adam . . . 32

4.1.8 AdaMax . . . 37

4.1.9 NAdam . . . 38

4.1.10 AMSGrad . . . 39

4.1.11 AdaBound . . . 40

4.1.12 RAdam . . . 41

4.1.13 Lookahead . . . 42
4.1.14 AdaMod . . . 43
4.2 Second-order Gradient Optimization Methods . . . 44
4.2.1 K-FAC . . . 44
4.2.2 Shampoo . . . 45
4.3 Additional Strategies for Optimizing SGD . . . 47
4.3.1 Decoupled Weight Decay (Not Accomplished) . . . 47
4.3.2 Gradient Noise . . . 49
4.3.3 Switching Optimizers . . . 50

II Machine Learning Algorithms **51**

5 Logistic Regression **52**
5.1 Vanilla Logistic Regression . . . 52
5.1.1 Notation . . . 52
5.1.2 Definition . . . 52
5.1.3 Learning . . . 52

6 Softmax Regression **54**
6.1 Vanilla Softmax Regression . . . 54
6.1.1 Definition . . . 54
6.1.2 Learning . . . 54
6.2 Variants of Softmax Regression . . . 55
6.2.1 Definition . . . 55
6.2.2 Learning . . . 55
6.3 Mixtape . . . 56
6.3.1 Definition . . . 56
6.3.2 Learning . . . 56

7 Naive Bayes **57**
7.1 Naive Bayes for Classification . . . 57
7.1.1 Definition . . . 57
7.1.2 Learning . . . 57
7.2 Naive Bayes for Regression . . . 57
7.2.1 Definition . . . 57
7.2.2 Learning . . . 57

8 Support Vector Machines (SVMs) **58**
8.1 Hard-Margin SVMs for Classification . . . 58
8.1.1 Definition . . . 58
8.1.2 Learning . . . 58

8.2 Soft-Margin SVMs for Classification . . . 59
8.2.1 Definition . . . 59
8.2.2 Learning . . . 59
8.2.3 Kernel Trick for Nonlinear Classification . . . 59

9 Decision Trees **60**
9.1 Classification & Regression Tree (CART) . . . 60
9.1.1 Definition . . . 60

10 Dimensionality Reduction **61**
10.1 Principal Component Analysis (PCA) . . . 61
10.1.1 Definition . . . 61
10.1.2 Learning . . . 61

III Ensemble Learning Algorithms **62**

11 Random Forest **63**

12 AdaBoost **64**
12.1 AdaBoost for Binary Classification . . . 64
12.2 AdaBoost for Multi Classification . . . 65
12.3 AdaBoost for Regression . . . 66

13 Gradient Boosting Machine (GBM) **67**
13.1 Vanilla Gradient Boosting Machine . . . 67
13.1.1 Definition . . . 67
13.1.2 Learning . . . 67

14 XGBoost **69**
14.1 Vanilla XGBoost . . . 69
14.1.1 Definition . . . 69
14.1.2 Learning . . . 69
14.1.3 Cognition . . . 73

15 LightGBM **74**

16 CatBoost **75**

IV Deep Learning Algorithms **76**

17 Feed-Forward Neural Networks (FFNNs) **77**

18 Convolutional Neural Networks (CNNs) **78**

19 Recurrent Neural Networks (RNNs) 79

20 Attention Mechanism 80
20.1 Vanilla Self-Attention . . . 80
20.1.1 Motivation . . . 80
20.1.2 Definition . . . 80
20.1.3 Cognition . . . 88
20.2 Bi-Directional Block Self-Attention . . . 89
20.2.1 Motivation . . . 89
20.2.2 Definition . . . 89
20.2.3 Cognition . . . 89
20.3 Multi-Head Attention . . . 90
20.3.1 Motivation . . . 90
20.3.2 Definition . . . 90
20.3.3 Cognition . . . 90

21 Encoder-Decoder Networks 91
21.1 Transformer . . . 91
21.1.1 Motivation . . . 91
21.1.2 Contribution . . . 91
21.1.3 Definition . . . 91
21.1.4 Cognition . . . 101
21.2 Transformer-XL . . . 102
21.2.1 Definition . . . 102
21.2.2 Cognition . . . 102
21.3 Reformers (Not Accomplished) . . . 103
21.3.1 Motivation . . . 103
21.3.2 Contribution . . . 103
21.3.3 Definition . . . 103
21.3.4 Cognition . . . 105

22 Generative Adversarial Networks (GANs) 106
22.1 Vanilla GANs . . . 106

23 Graph Neural Networks (GNNs) 107

24 Transfer Learning 108

25 Network Optimization and Regularization 109
25.1 Network Initialization . . . 109
25.2 Network Normalization . . . 110
25.2.1 Batch Normalization . . . 110

25.2.2 Layer Normalization . . . 113
25.2.3 Instance Normalization . . . 113
25.2.4 Group Normalization . . . 113
25.2.5 Batch-Instance Normalization . . . 113
25.3 Network Regularization . . . 113
25.3.1 Dropout . . . 113

V Reinforcement Learning Algorithms 114

26 Policy Gradient Methods 115
26.1 Vanilla Policy Gradient (VPG) . . . 115
26.1.1 Notation . . . 115
26.1.2 Definition . . . 115
26.1.3 Learning . . . 115
26.1.4 Variants of Return $\mathcal{G}(\tau; t)$ in Policy Gradient Methods (Not Accomplished) . . . 117
26.1.5 Cognition . . . 118

27 Value Approximation Methods 119
27.1 Q-Learning . . . 119

VI Algorithm Applications 120

28 Recommender System 121

29 Click-Through Rate Prediction 122
29.1 Factorization Machines (FM) . . . 122
29.1.1 Motivation . . . 122
29.1.2 Contribution . . . 122
29.1.3 Definition . . . 122
29.1.4 Learning . . . 122
29.1.5 Cognition . . . 122
29.2 Field-aware Factorization Machines for CTR Prediction (FFM) . . . 123
29.2.1 Motivation . . . 123
29.2.2 Contribution . . . 123
29.2.3 Definition . . . 123
29.2.4 Learning . . . 123
29.2.5 Cognition . . . 123
29.3 Automatic Feature Interaction Learning via Self-Attentive Neural Networks (AutoInt) . . . 124
29.3.1 Motivation . . . 124
29.3.2 Contribution . . . 124

29.3.3 Definition . . . 124
29.3.4 Cognition . . . 124
29.4 One-class Field-aware Factorization Machines for Recommender Systems with Implicit Feedbacks (OCFFM) . . . 125
29.4.1 Notation . . . 125
29.4.2 Motivation . . . 125
29.4.3 Contribution . . . 125
29.4.4 Definition . . . 125
29.4.5 Cognition . . . 126

30 Object Detection **127**
30.1 You Only Look Once: Unified, Real-Time Object Detection (YOLO) . . . 127
30.1.1 Notation . . . 127
30.1.2 Motivation . . . 127
30.1.3 Contribution . . . 127
30.1.4 Definition . . . 127
30.1.5 Continuation . . . 127
30.1.6 Cognition . . . 127
30.2 Single Shot MultiBox Detector (SSD) . . . 128
30.2.1 Motivation . . . 128
30.2.2 Contribution . . . 128
30.2.3 Definition . . . 128
30.2.4 Cognition . . . 128

31 Image Segmentation **129**

32 Image Super-Resolution **130**

33 Image Translation **131**
33.1 Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks (CycleGAN) . . . 131
33.1.1 Motivation . . . 131
33.1.2 Contribution . . . 131
33.1.3 Definition . . . 131
33.1.4 Cognition . . . 132
33.2 Unified Generative Adversarial Networks for Multi-Domain Image-to-Image Translation (StarGAN) . . . 133
33.2.1 Motivation . . . 133
33.2.2 Contribution . . . 133
33.2.3 Definition . . . 133
33.2.4 Cognition . . . 133

34 Pre-Trained Language Models **134**
34.1 Distributed Representations of Words and Phrases and their Compositionality (Word2Vec) . . . 134

34.1.1 Notation . . . 134
34.1.2 Motivation . . . 135
34.1.3 Contribution . . . 135
34.1.4 Definition (Not Accomplished) . . . 136
34.1.5 Cognition . . . 137
34.2 Global vectors for word representation (GloVe) . . . 138
34.2.1 Motivation . . . 138
34.2.2 Contribution . . . 138
34.2.3 Definition . . . 138
34.2.4 Cognition . . . 138
34.3 Deep Contextualized Word Representations (ELMo) . . . 139
34.3.1 Motivation . . . 139
34.3.2 Contribution . . . 139
34.3.3 Definition . . . 139
34.3.4 Cognition . . . 140
34.4 Improving Language Understanding by Generative Pre-Training (GPT) . . . 141
34.4.1 Motivation . . . 141
34.4.2 Contribution . . . 141
34.4.3 Definition . . . 141
34.5 Language Models are Unsupervised Multitask Learners (GPT-2) . . . 143
34.5.1 Motivation . . . 143
34.5.2 Contribution . . . 143
34.5.3 Definition . . . 143
34.5.4 Cognition . . . 143
34.6 Pre-training of Deep Bidirectional Encoder Representation from Transformers (BERT) . . . 144
34.6.1 Motivation . . . 144
34.6.2 Contribution . . . 144
34.6.3 Definition . . . 144
34.6.4 Continuation . . . 146
34.6.5 Cognition . . . 146
34.7 Improving Pre-training by Representing and Predicting Spans (SpanBERT) . . . 148
34.7.1 Motivation . . . 148
34.7.2 Contribution . . . 148
34.7.3 Definition . . . 148
34.7.4 Cognition . . . 149
34.8 Unified Language Model Pre-training for Natural Language Understanding and Generation (UniLM) . . . 150
34.8.1 Motivation . . . 150
34.8.2 Contribution . . . 150
34.8.3 Definition . . . 150

34.8.4 Cognition . . . 151
34.9 Generalized Autoregressive Pretraining for Language Understanding (XLNet) . . . 152
34.9.1 Motivation . . . 152
34.9.2 Contribution . . . 152
34.9.3 Definition (Not Accomplished) . . . 152
34.9.4 Cognition . . . 154
34.10 Retrieval-Augmented Language Model Pre-Training (REALM) . . . 155
34.10.1 Motivation . . . 155
34.10.2 Contribution . . . 155
34.10.3 Definition . . . 155
34.10.4 Cognition . . . 155

35 Semantic Matching **156**

36 Neural Machine Translation **157**

Appendices **158**

Appendix A Linear Algebra **159**
A.1 Matrices . . . 159
A.1.1 Matrix Product Operations . . . 159

Appendix B Tensor Calculus **161**
B.1 Differentiation of Univariate Functions . . . 161
B.2 Partial Differentiation and Gradients . . . 162

Appendix C Probability and Distributions **163**
C.1 Discrete Probability Distributions . . . 163
C.1.1 Total Variation Distance . . . 163
C.1.2 Hellinger Distance . . . 163
C.1.3 χ^2 and Kullback-Leibler Divergences . . . 163

Appendix D Stochastic Processes **164**
D.1 Markov Decision Processes (MDPs) . . . 164
D.1.1 Definition of MDP . . . 164
D.1.2 Policy & Objective . . . 164
D.1.3 Bellman Equations . . . 164
D.2 Partially Observable Markov Decision Processes (POMDPs) . . . 165

Appendix E Mathematical Optimization **166**

Appendix F Algorothm Design & Analysis **167**

Appendix G Programming Technology **168**

G.1 C++ 168

G.1.1 Streaming SIMD Extensions (SSE) 168

List of Figures

1 huber loss with $\delta = 1$ (green) and squared loss. Source: https://www.wikiwand.com/en/Huber_loss. 18

2 Triplet loss on two positive faces (Obama) and one negative face (Macron). Source: https://omoindrot.github.io/triplet-loss. . . . 23

3 The Triplet Loss minimizes the distance between an anchor and a positive, both of which have the same identity, and maximizes the distance between the anchor and a negative of a different identity. Source: (Schroff et al., 2015)[10]. . . . 24

4 The three types of negatives, given an anchor and a positive. Source: https://omoindrot.github.io/triplet-loss. . . . 24

5 Refine the definition of tree and define complexity of a tree. Source: https://homes.cs.washington.edu/\protect\unhbox\voidb@x\penalty\@M\{}tqchen/data/pdf/BoostedTree.pdf. . . . 71

6 The structure score calculation and efficient finding of the best split in XGBoost. Source: https://homes.cs.washington.edu/\protect\unhbox\voidb@x\penalty\@M\{}tqchen/data/pdf/BoostedTree.pdf. . . . 72

7 (left) Scaled dot-product attention. (right) Multi-head attention consists of several attention layers running in parallel. Source: (Vaswani et al., 2017)[11]. . . . 80

8 Parameter matrixs of linear mapping from $\mathcal{X}$ to $\mathcal{Q}$, $\mathcal{K}$ and $\mathcal{V}$ respectively. Note the distinction between dimension of original embedding space $d_{\mathcal{X}}$ and length of sequence N. . . . 81

9 Calculation process of the self-attention. Note that $d_{\mathcal{Q}} = d_{\mathcal{K}}$. . . . 81

10 Mapping functions from original word embedding space $\mathcal{X}$ to query embedding space $\mathcal{Q}$, key embedding space $\mathcal{K}$ and value embedding space $\mathcal{V}$. . . . 82

11 Self-Attention matrix operation interpretation. $\boldsymbol{Z} = \text{softmax}(\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}})\boldsymbol{V} \in \mathbb{R}^{N \times d_{\mathcal{V}}}$. . . . 84

12 Output sequence $\boldsymbol{Z}$ stacks N output vectors $\boldsymbol{z}^{(i)} \in \mathbb{R}^{d_{\mathcal{V}}}$ row by row. . . . 84

13 Self-attention operation of sequence "act gets results" with query word "action", which corresponds to $\left(\text{softmax}(\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}})\right)^{(1)}\boldsymbol{V} \in \mathbb{R}^{1 \times d_{\mathcal{V}}}$. . . . 84

14 Self-attention operation of sequence "act gets results" with query word "gets", which corresponds to $\left(\text{softmax}(\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}})\right)^{(2)}\boldsymbol{V} \in \mathbb{R}^{1 \times d_{\mathcal{V}}}$. . . . 85

15 Self-attention operation of sequence "act gets results" with query word "results", which corresponds to $\left(\text{softmax}(\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}})\right)^{(3)}\boldsymbol{V} \in \mathbb{R}^{1 \times d_{\mathcal{V}}}$. . . . 85

16 An example for interpreting masked self-attention mechanism. Source: https://jalammar.github.io/illustrated-gpt2/. . . . 86

17 An example for interpreting masked self-attention mechanism, computing $\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}}$ (before softmax operation). Source: https://jalammar.github.io/illustrated-gpt2/. . . . 86

18 An example for interpreting masked self-attention mechanism, get the masked socres matrix $\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}} + \boldsymbol{M} \in \mathbb{R}^{N \times N}$ based on the piror knowledge. Source: https://jalammar.github.io/illustrated-gpt2/. . . . 86

19 An example for interpreting the masked self-attention mechanism, get the masked socres matrix $\text{softmax}(\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}} + \boldsymbol{M}) \in \mathbb{R}^{N \times N}$ based on the piror knowledge. Source: https://jalammar.github.io/illustrated-gpt2/. . . . 87

20 The Transformer model architecture. Source: (Vaswani et al., 2017)[11]. . . . 91

21 Single-head self-attention. . . . 92

22 With multi-headed attention, we maintain separate $\boldsymbol{W}_{\mathrm{Q}}/\boldsymbol{W}_{\mathrm{K}}/\boldsymbol{W}_{\mathrm{V}}$ weight matrices for each head resulting in different $\boldsymbol{Q}/\boldsymbol{K}/\boldsymbol{V}$ matrices. As we did before, we multiply $\boldsymbol{X}$ by the different $\boldsymbol{W}_{\mathrm{Q}}/\boldsymbol{W}_{\mathrm{K}}/\boldsymbol{W}_{\mathrm{V}}$ matrices to produce different $\boldsymbol{Q}/\boldsymbol{K}/\boldsymbol{V}$ matrices. 93

23 With multi-headed attention, we calculate eight $\boldsymbol{Z}$ separately. 93

24 Concatenate multi attention heads $\boldsymbol{Z}^{(h)}, h \in \{1, \cdots, H\}$. 94

25 Trainable parameter matrix $\boldsymbol{W}_{\mathrm{O}} \in \mathbb{R}^{Hd_{\mathcal{V}} \times d_{\mathcal{Z}}}$ and the result matrix $\boldsymbol{Z} \in \mathbb{R}^{N \times d_{\mathcal{Z}}}$. Note always $d_{\mathcal{Z}} = d_{\mathcal{X}}$ for adding residual connection operation $\boldsymbol{X} + \boldsymbol{Z}$. 94

26 Multi-head self-attentions in one visual. The input embeddings can be either $\boldsymbol{X} \in \mathbb{R}^{N \times d_{\mathcal{X}}}$ (first layer) or $\boldsymbol{R}^{N \times d_{\mathcal{X}}}$ (start from second layer). Source: `http://jalammar.github.io/illustrated-transformer/`. 95

27 Residual connection & layer normalization after multi-head attention. $\tilde{\boldsymbol{Z}} = \mathrm{LayerNorm}(\boldsymbol{X} + \boldsymbol{Z}) \in \mathbb{R}^{N \times d_{\mathcal{X}}}$. 95

28 Visualize the first encoder layer of Transformer. 96

29 Connections between the last encoder and all decoders. Source: `http://jalammar.github.io/illustrated-transformer/`. 98

30 The expanded Transformer - model architecture. Note that $d_{\mathcal{X}} = d_{\mathcal{Y}}$ and $N_{\mathcal{X}} = N_{\mathcal{Y}}$ must be met. . . . 99

31 Embedding with positional encoding. Same embedding add with different positional encodings generate different result embeddings. Different embeddings add with same positional encoding generate different result embeddings. Note the positional encodings have the same dimension $d_{\mathcal{X}}$ as the embeddings, so that the two can be summed. 100

32 To give the model a sense of the order of the words, we add positional encoding vectors – the values of which follow a specific pattern. Source: `http://jalammar.github.io/illustrated-transformer/`. 100

33 A real example of positional encoding with a toy embedding size of 4. Source: `http://jalammar.github.io/illustrated-transformer/`. 101

34 **Normalization methods**. Each subplot shows a feature map tensor, with N as the batch axis, C as the channel axis, and(H, W) as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels. Source: (Wu et al., 2018)[12]. 110

35 Paired training data (left) consists of training examples $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{N}$, where the correspondence between $x^{(i)}$ and $y^{(i)}$ exists. We instead consider unpaired training data (right), consisting of a source set $\mathcal{D}_{\mathcal{X}} = \{x^{(i)}\}_{i=1}^{N_{\mathcal{X}}}$ and a target set $\mathcal{D}_{\mathcal{Y}} = \{y^{(j)}\}_{j=1}^{N_{\mathcal{Y}}}$ with no information provided as to which $x^{(i)}$ matches which $y^{(j)}$. Source: (Zhu et al., 2017)[13]. 131

36 (a) CyclaGAN (Zhu et al., 2017)[13] model contains two mapping functions (i.e., two generators) $G : X \mapsto Y$ and $F : Y \mapsto X$, and associated adversarial discriminators D_Y and D_X. D_Y encourages G to translate X into outputs indistinguishable from domain Y, and vice versa for D_X and F. To further regularize the mappings, we introduce two ***cycle consistency losses*** that capture the intuition that if we translate from one domain to the other and back again we should arrive at where we started: (b) forward cycle-consistency loss $x \mapsto G(x) \mapsto F(G(x)) \approx x$, and (c) backward cycle-consistency loss $y \mapsto F(y) \mapsto G(f(y)) \approx y$. Source: (Zhu et al., 2017)[13]. 131

37 Illustrations for a toy distributed representations $\boldsymbol{x}$ for all distinctive tokens in corpus $u \in \mathcal{D}$. Source: `https://www.adityathakker.com/introduction-to-word2vec-how-it-works/`. 135

38 Similar semantic meanings between (u_1, u_2) and (u_3, u_4) have similar embedding distances $\|\boldsymbol{x}_1 - \boldsymbol{x}_2\|$ and $\|\boldsymbol{x}_3 - \boldsymbol{x}_4\|$. Source: `https://algorithmia.com/algorithms/nlp/Word2Vec/docs`. 135

39 Example windows and process for computing $\Pr(u_{j+k}|u_j)$, i.e., Skip-gram method. 135

40 Example windows and process for computing $\Pr(u_j|u_{j+k})$, i.e., CBOW method. 136

41 ELMo uses the concatenation of independently trained left-to-right and right-to-left LSTMs to generate features for downstream tasks. Here $\boldsymbol{T}_j = \boldsymbol{h}_{j,L}^{\mathrm{LM}}$ represents the concatenation of the last forward encoding vector and backward encoding vector for a simplest case. Source: (Devlin et al., 2018)[14]. 140

42 An example for visualizing the $\boldsymbol{T}_j = \gamma^{\text{task}} \sum_{\ell=0}^{L} s_\ell^{\text{task}} \boldsymbol{h}_{j,\ell}^{\text{LM}}$, in this case we compute the embedding of token "stick" in sequence "Let's stick to" with 3 tokens. Source: `https://jalammar.github.io/illustrated-bert/`. 140

43 (left) Transformer architecture and training objectives used in this work. (right) Input transformations for fine-tuning on different tasks. We convert all structured inputs into token sequences to be processed by our pre-trained model, followed by a linear+softmax layer. Source: (Radford et al., 2018)[15]. 142

44 (left) Architecture of OpenAI GPT-2 (Ralford et al., 2018a)[16]. (right) OpenAI GPT (Ralford et al., 2018)[15] model. 143

45 Differences in pre-training model architectures. BERT uses a bidirectional Transformer (i.e., Transformer encoder-only blocks). OpenAI GPT uses a left-to-right Transformer (i.e., Transformer decoder-only blocks). ELMo uses the concatenation of independently trained left-to-right and right-toleft LSTMs to generate features for downstream tasks. Among the three, only BERT representations are jointly conditioned on both left and right context in all layers. In addition to the architecture differences, **BERT and OpenAI GPT are fine-tuning approaches, while ELMo is a feature-based approach**. Source: (Devlin et al., 2018)[14]. 144

46 BERT input representation. The input embeddings are the sum of the token embeddings, the segmentation embeddings and the position embeddings. Source: (Devlin et al., 2018)[14]. 145

47 Overall pre-training and fine-tuning procedures for BERT. Apart from output layers, the same architectures are used in both pre-training and fine-tuning. The same pre-trained model parameters are used to initialize models for different down-stream tasks. During fine-tuning, all parameters are fine-tuned. [CLS] is a special symbol added in front of every input example, and [SEP] is a special separator token (e.g. separating questions/answers). Source: (Devlin et al., 2018)[14]. 145

48 Illustrations of Fine-tuning BERT on Different Tasks. Source: (Devlin et al., 2018)[14]. 147

49 SpanBERT sample random span lengths from a geometric distribution $l \sim \text{Geo}(p = 0.2)$ clipped at $l_{\max} = 10$. which is skewed towards shorter spans. This yields a mean span length of $\text{mean}(l) = 3.8$. Source: (Joshi et al., 2019)[17]. 148

50 An illustration of SpanBERT training. The span "an American football game" is masked. The span boundary objective (SBO) uses the output representations of the boundary tokens $\boldsymbol{x}_4 \in \mathbb{R}^{d_{\mathcal{X}}}$ and $\boldsymbol{x}_9 \in \mathbb{R}^{d_{\mathcal{X}}}$ (in blue), to predict each token in the masked span. The equation shows the MLM and SBO objective terms for predict the token "football" (in pink), which as marked by the position embedding $\boldsymbol{p}_3 \in \mathbb{R}^{d_{\mathcal{X}}}$, is the 3-th token from $u_4 \in \{1, \cdots, n_{\mathcal{V}}\}$. Source: (Joshi et al., 2019)[17]. 149

51 Overview of unified LM pre-training. The model parameters are shared across the LM objectives (i.e., bidirectional LM, unidirectional LM, and sequence-to-sequence LM). We use different self-attention masks to control the access to context for each word token. The right-to-left LM is similar to the left-to-right one, which is omitted in the figure for brevity. Source: (Dong et al., 2019)[18]. 150

52 Illustration of the permutation language modeling objective for predicting u_3 ($\boldsymbol{x}_3$ as embedding) given the same input sequence $\mathcal{U}$ but with different factorization orders $\mathcal{I} \in \mathcal{Z}_{n_{\mathcal{U}}}$. ***In practise, we can get PLM by setting different mask matrix $\boldsymbol{M}_{\mathcal{I}}$***. Source: (Yang et al., 2019)[19]. 153

53 (a): Content stream attention, which is the same as the standard self-attention. (b): Query stream attention, which does not have access information about the content $u_{\mathcal{I}_j}$. (c): Overview of the permutation language modeling training with two-stream attention. Source: (Yang et al., 2019)[19]. 154

54 The perception-action-learning loop. At time t, the agent receives state s_t from the environment. The agent uses its policy to choose an action a_t. Once the action is executed, the environment transitions a step, providing the next state s_{t+1} as well as feedback in the form of a reward $\mathcal{R}_t$. The agent uses knowledge of state transitions, of the form $(s_t, a_t, s_{t+1}, \mathcal{R}_t)$, in order to learn and improve its policy. . . 164

List of Tables

1 Varients of $\mathcal{G}(\tau; t)$ in policy gradient method. 118
2 Notations used for the mathematical derivation of OCFFM in Subsection 29.4. 125
3 Notations used for the mathematical derivation of language models in Section 34. 134
4 Comparison between language model (LM) pre-training objectives. 150

Part I
Learning Algorithm Framework

1 Learning Algorithm Framework

In this section, we will introduce the framework of learning algorithm in both theoretical view in Subsection 1.1 and systems engineering view in Subsection 1.2.

1.1 Theoretical Learning Framework

In general, the goal of any supervised learning task is to get a map function $f : \mathbb{R}^{n_{\mathcal{X}}} \to \mathbb{R}$ (regression task) or $f : \mathbb{R}^{n_{\mathcal{X}}} \to \mathbb{R}^{n_C}$ (classification task), therefore the framework of supervised learning tasks, also some unsupervised learning tasks (e.g., generative adversarial task) can be universally divided into five steps:

- Dataset collection: $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{n_{\mathcal{D}}}$
- Inference construction: $y = f_\Theta(x) : \mathbb{R}^{n_{\mathcal{X}}} \to \mathbb{R}^{n_{\mathcal{Y}}}$
- Objective (Loss function & Regularization) selection: $\mathcal{J}(\Theta) = \mathcal{L}(\Theta) + \Omega(\Theta)$
- Optimizer selection: $\Theta \leftarrow \Theta + \Delta\Theta$
- Metrics performance: $\mathcal{E}\Big(\big(y^{(i)}, f_\Theta(x^{(i)})\big)\Big)$

1.2 System Engineering Learning Framework

2 Dataset

Each sample of a specific dataset $(x^{(i)}, y^{(i)}) \in \mathcal{D}$ consists of a series of independent variables $x_j^{(i)}, \forall j \in \{1, \cdots, |\mathcal{X}|\}$ and a dependent variable $y^{(i)}$. Content architecture of 2 shows following:

- Explanatory variable: $x_j^{(i)}$ 2.1
  - Categorical variable: one-hot encoding
  - Numerical variable: discretization
- Response variable: $y^{(i)}$ 2.2
  - Imbalanced label distribution

2.1 Explanatory Variable

2.1.1 Categorical Variable

One-hot encoding.

2.1.2 Numerical Variable

Discretization.

2.2 Response Variable

3 Objective Function

3.1 Loss Function for Regression

3.1.1 Squared Loss

Squared error as loss function shows following:

$$\mathcal{L}(y, f_\theta(x)) = (y - f_\theta(x))^2$$

And we can use pure mean squared error as the learning objective of model f_θ:

$$\min_\theta \ \mathcal{J}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \left(y^{(i)} - f_\theta(x^{(i)}) \right)^2$$

3.1.2 Absolute Loss

Absolute error as loss function shows following:

$$\mathcal{L}(y, f_\theta(x)) = |y - f_\theta(x)|$$

And we can use pure mean absolute error as the learning objective of model f_θ:

$$\min_\theta \ \mathcal{J}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \left| y^{(i)} - f_\theta(x^{(i)}) \right|$$

3.1.3 Huber Loss

Huber loss with hyper-parameter δ representing residual threshold as loss function shows following:

$$\mathcal{L}_\delta(y, h_\theta(x)) = \begin{cases} \frac{1}{2}(y - f_\theta(x))^2 & \text{if } |y - f_\theta(x)| \le \delta, \\ \delta \left| y - f_\theta(x) \right| - \frac{1}{2}\delta^2 & \text{otherwise.} \end{cases}$$

And we can use pure mean huber error as the learning objective of model f_θ:

$$\min_\theta \ \mathcal{J}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}_\delta(y^{(i)}, f_\theta(x^{(i)}))$$

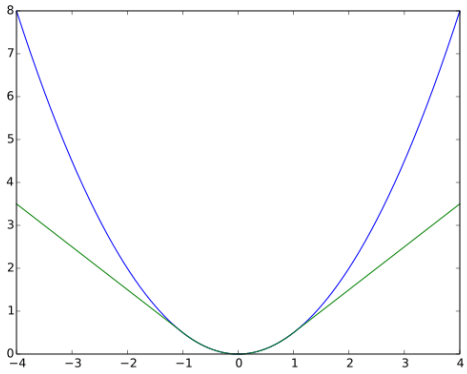


Figure 1: huber loss with $\delta = 1$ (green) and squared loss. Source: https://www.wikiwand.com/en/Huber_loss.

Compare the advantages and disadvantages among squared loss, absolute loss and huber loss:

- squared loss:
  - advantages:

    - The most common loss function for regression problem, differentiable everywhere.
    - pay more attention to the points $(y^{(i)}, f_\theta(x^{(i)}))$ with large residuals (errors) $\|y^{(i)} - f_\theta(x^{(i)})\|$ in the case of **non-outliers**.
  - disadvantages:
    - **Not robust**. Amplify the residuals of the outliers by square operation before a training iteration, therefore the objective pays too much attention to the outliers through punishment, fitting better for outliers, resulting in a large deviation in the model, so it is not robust enough.
- absolute loss:
  - advantages:
    - **Robust against outliers**. Therefore performs better than squared loss when there exists many outliers.
  - disadvantages:
    - Not differentiable when $y - f_\theta(x) = 0$ (although there almost never exists training residuals of 0 in general optimization procedures).
    - Worse fitting effect for **non-outlier large-residual** points than squared loss.
- huber loss:
  - advantages:
    - Robust against outliers, meanwhile better fitting non-outlier large-residual points.
    - Differentiable everywhere.

3.2 Loss Function for Classification

3.2.1 Hinge Loss

Hinge loss (most notably for SVMs) as loss function shows following:

$$\mathcal{L}(y, h_\theta(x)) = \max(0, 1 - y f_\theta(x))$$

Where:

- $y \in \{-1, +1\}$ represents the label output.
- $f_\theta(x) \in \mathbb{R}$ represents the predicted output. Inference as negative class when $f_\theta(x) < 0$, otherwise as positive class.

And we can use pure mean hinge loss as the learning objective of classification model f_θ:

$$\min_\theta \ \mathcal{J}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \max\left(0, 1 - y^{(i)} f_\theta(x^{(i)})\right)$$

Hinge loss results in a clear inter-class effect and a blurred intra-class effect.

3.2.2 Log-Likelihood

Assuming that all samples $(x^{(i)}, y^{(i)}) \in \mathcal{D}$ satisfy the assumption of independent and identical distribution, maximizing the joint probability of all training observation samples, we can get the pattern relationship (a map function $f : \mathcal{X} \to \mathcal{Y}$) between independent variable $x \in \mathcal{X}$ and dependent variable $y \in \mathcal{Y}$.

$$\begin{aligned} \max \ \Pr(\mathcal{D}) &= \Pr(\{(x^{(i)}, y^{(i)})\}_{i=1}^{N}) \\ &= \prod_{i=1}^{N} \Pr(x^{(i)}, y^{(i)}) \quad (\because \text{i.i.d. of samples in } \mathcal{D}) \end{aligned}$$

If we impose a logarithmic operation for above objective function, it will not change the monotonicity of above objective. Besides, considering the probability density of each observation $(x^{(i)}, y^{(i)})$, namely $\frac{1}{N}$, we have a likelihood objective as follows:

$$\begin{aligned} \max \ \mathcal{J} &= \frac{1}{N} \log \Pr(\mathcal{D}) \\ &= \frac{1}{N} \log \left(\prod_{i=1}^{N} \Pr(x^{(i)}, y^{(i)}) \right) \\ &= \frac{1}{N} \sum_{i=1}^{N} \log \Pr(x^{(i)}, y^{(i)}) \\ &= \frac{1}{N} \sum_{i=1}^{N} \log \left(\Pr(y^{(i)} | x^{(i)}) \Pr(x^{(i)}) \right) \quad \text{(conditional probability)} \\ &= \frac{1}{N} \sum_{i=1}^{N} \left(\log \Pr(y^{(i)} | x^{(i)}) + \log \Pr(x^{(i)}) \right) \\ &= \frac{1}{N} \sum_{i=1}^{N} \log \Pr(y^{(i)} | x^{(i)}) + \frac{1}{N} \sum_{i=1}^{N} \log \Pr(x^{(i)}) \end{aligned}$$

we can't estimate the priori probability $\Pr(x^{(i)})$ here, or $\Pr(x^{(i)}) \approx \frac{1}{N}$. If the conditional probability function $\Pr(y|\boldsymbol{x}; \theta)$ is estimated by parameters θ, therefore we have the objective:

$$\max_\theta \ \mathcal{J}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \log \Pr(y^{(i)} | x^{(i)}; \theta)$$

3.2.3 Kullback-Leibler Divergence

Perform an equivalent conversion from **maximizing log-likelihood** objective to **minimizing kl-divergence** objective:

$$\max \ \frac{1}{N}\sum_{i=1}^{N}\log \Pr(y^{(i)}|x^{(i)};\theta) \quad \text{(Target observation is sampled from a prob distribution: } y^{(i)} \sim p^{(i)} \in \mathbb{P}^{|\mathcal{Y}|})$$

$$\Leftrightarrow \min \ \frac{1}{N}\sum_{i=1}^{N}\mathrm{KL}(p^{(i)}\|f_\theta(x^{(i)})) \quad \text{(Predicted prob distribution: } q^{(i)} = f_\theta(x^{(i)}) \in \mathbb{P}^{|\mathcal{Y}|})$$

$$\Rightarrow \min \ \frac{1}{N}\sum_{i=1}^{N}\mathrm{KL}(p^{(i)}\|q^{(i)})$$

$$\Rightarrow \min \ \frac{1}{N}\sum_{i=1}^{N}\left(p^{(i)}\cdot\log\frac{p^{(i)}}{q^{(i)}}\right) \quad \text{(Definition of Kullback-Leibler divergence, both } \frac{\cdot}{\cdot} \text{ and } \log(\cdot) \text{ are element-wise operations)}$$

$$\Rightarrow \min \ \frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K}p_k^{(i)}\log\frac{p_k^{(i)}}{q_k^{(i)}}$$

$$\Rightarrow \min \ \frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K}\left(p_k^{(i)}\log p_k^{(i)} - p_k^{(i)}\log q_k^{(i)}\right) \quad (K = |\mathcal{Y}| \text{ is the total number of classes})$$

$$\Rightarrow \min \ \underbrace{\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K}p_k^{(i)}\log p_k^{(i)}}_{\text{entropy}} \underbrace{-\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K}p_k^{(i)}\log q_k^{(i)}}_{\text{cross entropy}}$$

$$\Rightarrow \min_\theta \ \mathcal{J}(\theta) = \underbrace{\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K}p_k^{(i)}\log p_k^{(i)}}_{\text{constant}} - \frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K}p_k^{(i)}\log\left[f_\theta(x^{(i)})\right]_k$$

Where:

- $p \in \mathbb{P}^K$: target probability distribution (often 0-1 probability distribution) over K classes
- $q = f_\theta(x) \in \mathbb{P}^K$: predicted probability distribution over K classes:

$$f_\theta(x) = [\Pr(\text{class}(x) = 1), \Pr(\text{class}(x) = 2), \ldots, \Pr(\text{class}(x) = K)]\,.$$

3.2.4 Cross Entropy

Definition of Cross entropy between the distributions $p \in \mathbb{P}^K$ and $q \in \mathbb{P}^K$ is as follow:

$$\begin{aligned}\mathcal{H}(p,q) &= \mathbb{E}_p[-\log q] \\ &= -p\cdot\log q \\ &= -\sum_{k=1}^{K}p_k\log q_k \\ &= -\langle p, \log q\rangle\end{aligned}$$

where the inner product $<\cdot,\cdot>$ computes a similarity measure between the network predicted probability distribution $\boldsymbol{q}$ and the corresponding data label probability distribution $\boldsymbol{p}$. Therefore the pure cross entropy as objective shows following:

$$\min_\theta \ \mathcal{J}(\theta) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K}p_k^{(i)}\log\left[f_\theta(x^{(i)})\right]_k$$

Comprehension. **maximizing the cosine similarity** between vectors (representing discrete probability distribution) p and $\log q$, is equivalent to **minimizing cross entropy** $\mathcal{H}(p,q)$.

3.2.5 Focal Loss

Definition of Focal Loss[20] for a training sample (x, y) shows following:

$$\mathcal{L}_{\text{FL}} = -\sum_{k=1}^{K} \eta_k (1 - \tilde{q}_k)^{\gamma} p_k \log q_k$$

Where:

- p_k: target probability of k-th class, the ground-truth target y is sampled from target probability distribution over classes $y \sim p \in \mathbb{P}^K$.
- q_k: predicted probability of k-th class, namely $q_k = [f_\theta(x)]_k$.
- $\tilde{q}_k$: degree of correctness of k-th class, namely:

  $$\tilde{q}_k = \begin{cases} q_k & \text{if } y = k \text{ (positive class)}, \\ 1 - q_k & \text{otherwise (negative class)}. \end{cases}$$

  therefore the calculation of $\tilde{q}_k$ depends on not only indepentent variable x (probability inference q_k indeed), but also corresponding ground-truth target y.
- $\eta_k \in [0, 1]$: (hyper-parameter) a weighting factor **for balancing frequent class (e.g., negative class) and rare class (e.g., positive class)**. In practice η_k may be set by inverse class frequency or treated as a hyperparameter to set by cross validation.
- $\gamma \geq 0$: (hyper-parameter) a focusing factor **for hard sample mining** (i.e., $\tilde{q}_k \leq 0.5$). For instance, with $\gamma = 2$ an easy sample with $\tilde{q}_k = 0.9$ would have 100 times ($\because (1 - \tilde{q}_k)^{\gamma} = 0.01$) lower loss compared with cross entropy, another easy sample with $\tilde{q}_k = 0.968$ would have 1000 times ($\because (1 - \tilde{q}_k)^{\gamma} \approx 0.001$) lower loss compared with cross entropy. The larger the γ value, the stronger the focusing strength compared with cross entropy.

Therefore the objective over whole training samples can be formulated as follow:

$$\min_{\theta} \mathcal{J}(\theta) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{K} \eta_k (1 - \tilde{q}_k)^{\gamma} p_k \log q_k$$

Intuition. Consider the case that **negative and easy samples overwhelm during training process**:

- **Negative samples overwhelming case (Imbalance case).** When there exists extreme imbalance between positive class and negative classes during training (e.g., $1 : 1000$), we can hardly learn from samples with positive class (because rare positive samples) using cross entropy as loss function.

$$\begin{aligned} \mathcal{J}(\theta) &= -\frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{K} p_k^{(i)} \log q_k^{(i)} \\ &= \frac{1}{N_{\text{pos}}} \sum_{i=1}^{N_{\text{pos}}} \left(-\sum_{k=1}^{K} p_k^{(i)} \log q_k^{(i)} \right) + \frac{1}{N_{\text{neg}}} \sum_{i=1}^{N_{\text{neg}}} \left(-\sum_{k=1}^{K} p_k^{(i)} \log q_k^{(i)} \right) \quad (N_{\text{pos}} \ll N_{\text{neg}}) \\ &\approx -\frac{1}{N_{\text{neg}}} \sum_{i=1}^{N_{\text{neg}}} \sum_{k=1}^{K} p_k^{(i)} \log q_k^{(i)} \end{aligned}$$

- **Easy sample overwhelming case.** we can't learn the important points from easy samples.

Therefore the most disadvantage of cross entropy for training set overwhelmed by negative and easy samples is, the learning effect of the positive samples during the training phase is poor, which leads to a very low prediction accuracy of the positive samples during the prediction phase.

Reading Materials.

- Focal Loss for Dense Object Detection[20]
- Li Mao's Log Book: Use Focal Loss To Train Model Using Imbalanced Dataset

3.2.6 Triplet Loss

Definition of Triplet Loss[21][10] for a training tuple $(x_{\mathrm{A}}, x_{\mathrm{P}}, x_{\mathrm{N}})$ (represent anchor, positive and negative respectively) shows following:

$$\mathcal{L}\big(f_\theta(x_{\mathrm{A}}), f_\theta(x_{\mathrm{P}}), f_\theta(x_{\mathrm{N}})\big) = \max\left(\|f_\theta(x_{\mathrm{A}}) - f_\theta(x_{\mathrm{P}})\|_2^2 - \|f_\theta(x_{\mathrm{A}}) - f_\theta(x_{\mathrm{N}})\|_2^2 + \alpha, 0\right)$$

Where $f_\theta(x) \in \mathbb{R}^n$ represents the embedding of sample $x \in \mathbb{R}^d$, additionally, **we constrain this embedding to live on the n-dimensional unit hypersphere, i.e., $\|f_\theta(x)\| = 1$**; $\alpha \geq 0$ is a hyper-parameter representing the margin.

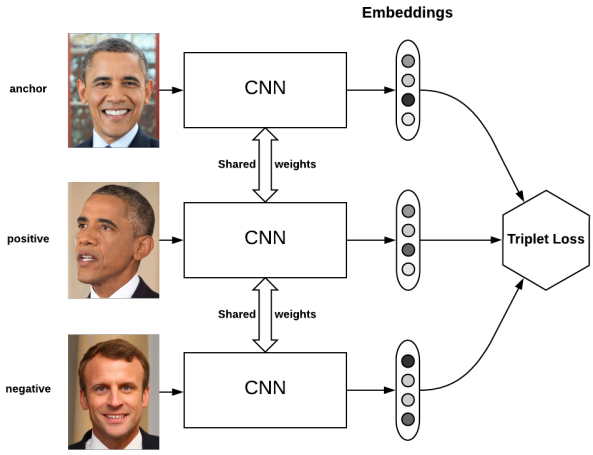


Figure 2: Triplet loss on two positive faces (Obama) and one negative face (Macron). Source: https://omoindrot.github.io/triplet-loss.

Therefore, we have the objective with triplet loss as follow:

$$\min_\theta \ \mathcal{J}(\theta) = \frac{1}{N}\sum_{i=1}^{N} \max\left(\|f_\theta(x_{\mathrm{A}}) - f_\theta(x_{\mathrm{P}})\|_2^2 - \|f_\theta(x_{\mathrm{A}}) - f_\theta(x_{\mathrm{N}})\|_2^2 + \alpha, 0\right)$$

Implementation: TensorFlow (A Naive Version).

```
# Copyright (C) 2019 Tong Jia. All rights reserved.
"""Definitions of Triplet loss, a naive version."""
import tensorflow as tf


margin = 0.2

anchor_embed = ...    # shape [None, 128]
positive_embed = ...  # shape [None, 128]
negative_embed = ...  # shape [None, 128]

d_pos = tf.reduce_sum(tf.square(anchor_embed - positive_embed), 1)  # shape [None]
d_neg = tf.reduce_sum(tf.square(anchor_embed - negative_embed), 1)  # shape [None]

loss = tf.maximum(0.0, margin + d_pos - d_neg)  # shape [None]
```

Comprehension.

- $\min_\theta \ \|f_\theta(x_{\mathrm{A}}) - f_\theta(x_{\mathrm{P}})\|_2^2$: Minimize the distance between anchor point $f_\theta(x_{\mathrm{A}})$ and corresponding positive point $f_\theta(x_{\mathrm{P}})$.
- $\max_\theta \ \|f_\theta(x_{\mathrm{A}}) - f_\theta(x_{\mathrm{N}})\|_2^2$: Maximize the distance between anchor point $f_\theta(x_{\mathrm{A}})$ and corresponding negative point $f_\theta(x_{\mathrm{N}})$. Equivalent to: $\min_\theta \ -\|f_\theta(x_{\mathrm{A}}) - f_\theta(x_{\mathrm{N}})\|_2^2$.

- $\min_{\theta} \|f_\theta(x_\text{A}) - f_\theta(x_\text{P})\|_2^2 - \|f_\theta(x_\text{A}) - f_\theta(x_\text{N})\|_2^2 + \alpha$: Promote the distance between anchor point $f_\theta(x_\text{A})$ and corresponding positive point $f_\theta(x_\text{P})$ is relatively smaller than the distance between anchor point $f_\theta(x_\text{A})$ and corresponding negative point $f_\theta(x_\text{N})$. **What we expect** is:

$$
\begin{aligned}
&\|f_\theta(x_\text{A}) - f_\theta(x_\text{P})\|_2^2 \leq \|f_\theta(x_\text{A}) - f_\theta(x_\text{N})\|_2^2 \\
\Rightarrow\ &\|f_\theta(x_\text{A}) - f_\theta(x_\text{P})\|_2^2 - \|f_\theta(x_\text{A}) - f_\theta(x_\text{N})\|_2^2 \leq 0 \\
\Rightarrow\ &\|f_\theta(x_\text{A}) - f_\theta(x_\text{N})\|_2^2 - \|f_\theta(x_\text{A}) - f_\theta(x_\text{P})\|_2^2 \geq 0 \\
\Rightarrow\ &\|f_\theta(x_\text{A}) - f_\theta(x_\text{N})\|_2^2 - \|f_\theta(x_\text{A}) - f_\theta(x_\text{P})\|_2^2 \geq 0 + \alpha \quad (\alpha \geq 0 \text{ is the margin, a stricter restriction}) \\
\Rightarrow\ &\|f_\theta(x_\text{A}) - f_\theta(x_\text{P})\|_2^2 - \|f_\theta(x_\text{A}) - f_\theta(x_\text{N})\|_2^2 + \alpha \leq 0
\end{aligned}
$$

- $\max\left(\|f_\theta(x_\text{A}) - f_\theta(x_\text{P})\|_2^2 - \|f_\theta(x_\text{A}) - f_\theta(x_\text{N})\|_2^2 + \alpha, 0\right)$:
  - If the restriction (expectation) $\|f_\theta(x_\text{A}) - f_\theta(x_\text{N})\|_2^2 - \|f_\theta(x_\text{A}) - f_\theta(x_\text{P})\|_2^2 \geq \alpha$ is met: namely $\|f_\theta(x_\text{A}) - f_\theta(x_\text{P})\|_2^2 - \|f_\theta(x_\text{A}) - f_\theta(x_\text{N})\|_2^2 + \alpha \leq 0$ is met, it means we have a just suitable embedding function $f_\theta : \mathbb{R}^d \to \mathbb{R}^n$ for tuple $(x_\text{A}, x_\text{P}, x_\text{N})$, therefore we don't need to optimize f_θ when using training tuple $(x_\text{A}, x_\text{P}, x_\text{N})$.
  - On the contrary, if here we have a result $\|f_\theta(x_\text{A}) - f_\theta(x_\text{N})\|_2^2 - \|f_\theta(x_\text{A}) - f_\theta(x_\text{P})\|_2^2 < \alpha$: namely $\|f_\theta(x_\text{A}) - f_\theta(x_\text{P})\|_2^2 - \|f_\theta(x_\text{A}) - f_\theta(x_\text{N})\|_2^2 + \alpha > 0$, At this time, the effect that the distance between anchor point $f_\theta(x_\text{A})$ and corresponding positive point $f_\theta(x_\text{P})$ is sufficiently smaller than the distance between the anchor point $f_\theta(x_\text{A})$ and corresponding negative point $f_\theta(x_\text{N})$ **has not been achieved** (See 3 before learning).

  Therefore we need to optimize f_θ in order to achieve expected effect. Combine the above two situations, we can get the triplet loss:

$$\mathcal{L}\big(f_\theta(x_\text{A}), f_\theta(x_\text{P}), f_\theta(x_\text{N})\big) = \max\left(\|f_\theta(x_\text{A}) - f_\theta(x_\text{P})\|_2^2 - \|f_\theta(x_\text{A}) - f_\theta(x_\text{N})\|_2^2 + \alpha, 0\right)$$

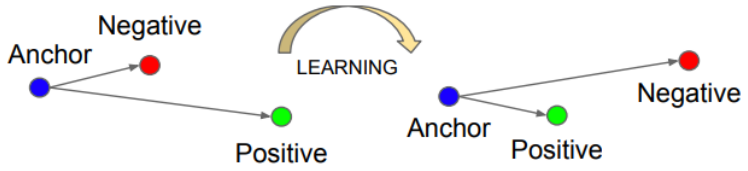


Figure 3: The Triplet Loss minimizes the distance between an anchor and a positive, both of which have the same identity, and maximizes the distance between the anchor and a negative of a different identity. Source: (Schroff et al., 2015)[10].

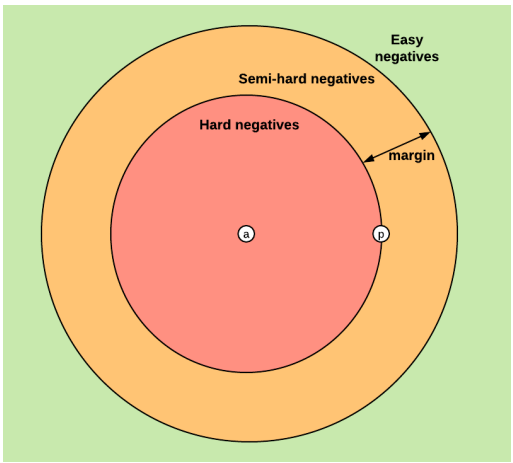


Figure 4: The three types of negatives, given an anchor and a positive. Source: https://omoindrot.github.io/triplet-loss.

Strategies for Triplet Mining. There exists three types of negatives x_N, given an anchor x_A and a positive x_P:

- Hard negative: $\|x_{\mathrm{A}} - x_{\mathrm{N}}\| \leq \|x_{\mathrm{A}} - x_{\mathrm{P}}\|$
- Semi-hard negative: $\|x_{\mathrm{A}} - x_{\mathrm{P}}\| < \|x_{\mathrm{A}} - x_{\mathrm{N}}\| \leq \|x_{\mathrm{A}} - x_{\mathrm{P}}\| + \alpha$
- Easy negative: $\|x_{\mathrm{A}} - x_{\mathrm{N}}\| > \|x_{\mathrm{A}} - x_{\mathrm{P}}\| + \alpha$

In paper FaceNet[10], author constructed a tuple by picking a random semi-hard negative for a given anchor and positive, then train on these triplets.

Cognition.

1. Annealing strategy for margin α (more and more strict).
2. Different strategies for hard negatives or semi-hard negatives mining.

Reading Materials.

- Deep Metric Learning Using Triplet Network[21].
- FaceNet: A Unified Embedding for Face Recognition and Clustering[10].
- Deep Metric Learning with Hierarchical Triplet Loss[22].
- Olivier Moindrot blog: Triplet Loss and Online Triplet Mining in TensorFlow.
- Code: Implementation of triplet loss in TensorFlow.

4 Optimization Algorithm

4.1 First-order Gradient Optimization Methods

4.1.1 SGD

Algorithm 1: SGD algorithm for stochastic optimization[23].

Input: $\{\alpha_t \in \mathbb{R}\}_{t=1}^{T}$: Stepsize of each iteration;
$\theta_0 \in \mathbb{R}^n$: Initial parameters ($n$ is the number of parameters).
Output: $\theta_T \in \mathbb{R}^n$: Resulting parameters.

for $t \in \{1, \ldots, T\}$ **do**
  $\{(x^{(i)}, y^{(i)})\}_{i=1}^{M} \sim \mathcal{D}$ ▷ Sample a minibatch of M samples with corresponding targets from training set $\mathcal{D}$
  $g_t \leftarrow \frac{1}{M} \sum_{i=1}^{M} \nabla_\theta \mathcal{L}\left(f(x^{(i)}; \theta_{t-1}), y^{(i)}\right)$ ▷ Calculate gradients w.r.t. stochastic objective at timestep t
  $\theta_t \leftarrow \theta_{t-1} - \alpha_t g_t$ ▷ Update parameters
end
return θ_T

The selection of step sizes $\{\alpha_t\}_{t=1}^{T}$ usually adopts some annealing strategy to prevent gradient explosion in the late training.

Implementation: PyTorch.

Reading Materials.

- Sebastian Ruder: An overview of gradient descent optimization algorithms[24].

4.1.2 Momentum

Algorithm 2: SGD with momentum algorithm for stochastic optimization[25].

Input: $\{\alpha_t \in \mathbb{R}\}_{t=1}^{T}$: Stepsize of each iteration;
$\beta \in \mathbb{R}$: Exponential decay rate to calculate moving 1st moment;
$m_0 \in \mathbb{R}^n$: Initial moving 1st moment, perhaps 0 at very beginning;
$\theta_0 \in \mathbb{R}^n$: Initial parameters ($n$ is the number of parameters).
Output: $\theta_T \in \mathbb{R}^n$: Resulting parameters.

for $t \in \{1, \ldots, T\}$ **do**
  $\{(x^{(i)}, y^{(i)})\}_{i=1}^{M} \sim \mathcal{D}$ ▷ Sample a minibatch of M samples with corresponding targets from training set $\mathcal{D}$
  $g_t \leftarrow \frac{1}{M} \sum_{i=1}^{M} \nabla_\theta \mathcal{L}\left(f(x^{(i)}; \theta_{t-1}), y^{(i)}\right)$ ▷ Calculate gradients w.r.t. stochastic objective at timestep t
  $m_t \leftarrow \beta m_{t-1} + \alpha_t g_t$ ▷ Update exponential moving 1st moment
  $\theta_t \leftarrow \theta_{t-1} - m_t$ ▷ Update parameters
end
return θ_T

4.1.3 Nesterov Momentum

Algorithm 3: Nesterov accelerated gradient (NAG) algorithm for stochastic optimization[24].

Input: $\{\alpha_t \in \mathbb{R}\}_{t=1}^{T}$: Stepsize of each iteration;
$\beta \in \mathbb{R}$: Exponential decay rate to calculate moving 1st moment;
$m_0 \in \mathbb{R}^n$: Initial moving 1st moment, perhaps 0 at very beginning;
$\theta_0 \in \mathbb{R}^n$: Initial parameters ($n$ is the number of parameters).
Output: $\theta_T \in \mathbb{R}^n$: Resulting parameters.

for $t \in \{1, \ldots, T\}$ **do**
  $\{(x^{(i)}, y^{(i)})\}_{i=1}^{M} \sim \mathcal{D}$ ▷ Sample a minibatch of M samples with corresponding targets from training set $\mathcal{D}$
  $\tilde{\theta}_{t-1} \leftarrow \theta_{t-1} - \beta m_{t-1}$ ▷ Update interimly
  $g_t \leftarrow \frac{1}{M} \sum_{i=1}^{M} \nabla_\theta \mathcal{L}\left(f(x^{(i)}; \tilde{\theta}_{t-1}), y^{(i)}\right)$ ▷ Calculate gradients w.r.t. stochastic objective at timestep t
  $m_t \leftarrow \beta m_{t-1} + \alpha_t g_t$ ▷ Update exponential moving 1st moment
  $\theta_t \leftarrow \theta_{t-1} - m_t$ ▷ Update parameters
end
return θ_T

Reading Materials.

- jlmelville: Nesterov Accelerated Gradient and Momentum.

4.1.4 AdaGrad

Algorithm 4: AdaGrad algorithm for stochastic optimization[26].

Input: $\{\alpha_t \in \mathbb{R}\}_{t=1}^{T}$: Stepsize of each iteration;
$\epsilon \in \mathbb{R}$: Small constant, perhaps 10^{-7} for numerical stability;
$v_0 \in \mathbb{R}^n$: Initial moving 2nd moment, perhaps 0 at very beginning;
$\theta_0 \in \mathbb{R}^n$: Initial parameters ($n$ is the number of parameters).
Output: $\theta_T \in \mathbb{R}^n$: Resulting parameters.

for $t \in \{1, \ldots, T\}$ **do**
  $\{(x^{(i)}, y^{(i)})\}_{i=1}^{M} \sim \mathcal{D}$ ▷ Sample a minibatch of M samples with corresponding targets from training set $\mathcal{D}$
  $g_t \leftarrow \frac{1}{M} \sum_{i=1}^{M} \nabla_\theta \mathcal{L}\left(f(x^{(i)}; \theta_{t-1}), y^{(i)}\right)$ ▷ Calculate gradients w.r.t. stochastic objective at timestep t
  $v_t \leftarrow v_{t-1} + g_t \odot g_t$ ▷ Update accumulated moving 2nd moment
  $\theta_t \leftarrow \theta_{t-1} - \frac{\alpha_t}{\sqrt{v_t + \epsilon}} \odot g_t$ ▷ Update parameters, division and root operation applied element-wise
end
return θ_T

4.1.5 RMSprop

Algorithm 5: RMSProp algorithm for stochastic optimization[27].

Input: $\{\alpha_t \in \mathbb{R}\}_{t=1}^{T}$: Stepsize of each iteration;
$\beta \in \mathbb{R}$: Exponential decay rate to calculate moving 2nd moment;
$\epsilon \in \mathbb{R}$: Small constant, perhaps 10^{-8} for numerical stability;
$v_0 \in \mathbb{R}^n$: Initial moving 2nd moment, perhaps 0 at very beginning;
$\theta_0 \in \mathbb{R}^n$: Initial parameters ($n$ is the number of parameters).

Output: $\theta_T \in \mathbb{R}^n$: Resulting parameters.

for $t \in \{1, \ldots, T\}$ **do**
  $\{(x^{(i)}, y^{(i)})\}_{i=1}^{M} \sim \mathcal{D}$ ▷ Sample a minibatch of M samples with corresponding targets from training set $\mathcal{D}$
  $g_t \leftarrow \frac{1}{M} \sum_{i=1}^{M} \nabla_\theta \mathcal{L}\left(f(x^{(i)}; \theta_{t-1}), y^{(i)}\right)$ ▷ Calculate gradients w.r.t. stochastic objective at timestep t
  $v_t \leftarrow \beta v_{t-1} + (1-\beta) g_t \odot g_t$ ▷ Update exponential moving 2nd moment
  $\theta_t \leftarrow \theta_{t-1} - \frac{\alpha_t}{\sqrt{v_t + \epsilon}} \odot g_t$ ▷ Update parameters, division and root operation applied element-wise
end
return θ_T

The only difference between AdaGrad and RMSProp is that AdaGrad uses accumulated moving 2nd moment:

$$v_t \leftarrow v_{t-1} + g_t \odot g_t$$

but RMSProp uses exponential moving 2nd moment:

$$v_t \leftarrow \beta v_{t-1} + (1-\beta) g_t \odot g_t$$

4.1.6 AdaDelta

Algorithm 6: AdaDelta algorithm for stochastic optimization[28].

Input: $\{\alpha_t \in \mathbb{R}\}_{t=1}^{T}$: Stepsize of each iteration;
$\rho \in [0, 1)$: Decay rates, perhaps 0.95;
$\epsilon \in \mathbb{R}$: Small constant, perhaps 10^{-7} for numerical stability;
$\mathbb{E}[g^2]_0 \in \mathbb{R}^n$: Initial accumulation of squared gradient, perhaps 0 at very beginning;
$\mathbb{E}[\Delta\theta^2]_0 \in \mathbb{R}^n$: Initial accumulation of squared update, perhaps 0 at very beginning;
$\theta_0 \in \mathbb{R}^n$: Initial parameters ($n$ is the number of parameters).

Output: $\theta_T \in \mathbb{R}^n$: Resulting parameters.

for $t \in \{1, \ldots, T\}$ **do**
- $\{(x^{(i)}, y^{(i)})\}_{i=1}^{M} \sim \mathcal{D}$ ▷ Sample a minibatch of M samples with corresponding targets from training set $\mathcal{D}$
- $g_t \leftarrow \frac{1}{M} \sum_{i=1}^{M} \nabla_\theta \mathcal{L}\left(f(x^{(i)}; \theta_{t-1}), y^{(i)}\right)$ ▷ Calculate gradients w.r.t. stochastic objective at timestep t
- $\mathbb{E}[g^2]_t \leftarrow \rho \mathbb{E}[g^2]_{t-1} + (1-\rho) g_t^2$ ▷ Accumulate squared gradient
- $\Delta\theta_t \leftarrow -\alpha_t \frac{\sqrt{\mathbb{E}[\Delta\theta^2]_{t-1} + \epsilon}}{\sqrt{\mathbb{E}[g^2]_t + \epsilon}} g_t$ ▷ Compute update
- $\mathbb{E}[\Delta\theta^2]_t \leftarrow \rho \mathbb{E}[\Delta\theta^2]_{t-1} + (1-\rho) \Delta\theta_t^2$ ▷ Accumulate squared update
- $\theta_t \leftarrow \theta_{t-1} + \Delta x_t$ ▷ Update parameters

end

return θ_T

4.1.7 Adam

Algorithm 7: Adam algorithm for stochastic optimization[29].

Input: $\{\alpha_t \in \mathbb{R}\}_{t=1}^{T}$: Stepsize of each iteration;
$\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates to calculate 1st and 2nd moment, perhaps $(0.9, 0.999)$;
$\epsilon \in \mathbb{R}$: Small constant, perhaps 10^{-8} for numerical stability;
$m_0 \in \mathbb{R}^n$: Initial moving 1st moment, perhaps 0 at very beginning;
$v_0 \in \mathbb{R}^n$: Initial moving 2nd moment, perhaps 0 at very beginning;
$\theta_0 \in \mathbb{R}^n$: Initial parameters ($n$ is the number of parameters).

Output: $\theta_T \in \mathbb{R}^n$: Resulting parameters.

for $t \in \{1, \ldots, T\}$ **do**
- $\{(x^{(i)}, y^{(i)})\}_{i=1}^{M} \sim \mathcal{D}$ ▷ Sample a minibatch of M samples with corresponding targets from training set $\mathcal{D}$
- $g_t \leftarrow \frac{1}{M} \sum_{i=1}^{M} \nabla_\theta \mathcal{L}\left(f(x^{(i)}; \theta_{t-1}), y^{(i)}\right)$ ▷ Calculate gradients w.r.t. stochastic objective at timestep t
- $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$ ▷ Update biased 1st moment estimate
- $v_t \leftarrow \beta v_{t-1} + (1 - \beta) g_t \odot g_t$ ▷ Update biased 2nd moment estimate
- $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ ▷ Compute bias-corrected 1st moment estimate
- $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ ▷ Compute bias-corrected 2nd moment estimate
- $\eta_t \leftarrow \alpha_t / (\sqrt{\widehat{v}_t} + \epsilon)$ ▷ Compute element-wise adaptive stepsizes
- $\theta_t \leftarrow \theta_{t-1} - \eta_t \odot \widehat{m}_t$ ▷ Update parameters

end

return θ_T ▷ Resulting parameters

In practise (implementation using TensorFlow or PyTorch), we can write an use update as follow:

$$
\begin{aligned}
\theta_t &\leftarrow \theta_{t-1} - \alpha_t \widehat{m}_t / (\sqrt{\widehat{v}_t}) \\
&\leftarrow \theta_{t-1} - \alpha_t \frac{\frac{m_t}{1-\beta_1^t}}{\sqrt{\frac{v_t}{1-\beta_2^t}}} \\
&\leftarrow \theta_{t-1} - \alpha_t \frac{m_t}{\sqrt{v_t}} \odot \frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t} \\
&\leftarrow \theta_{t-1} - \frac{\alpha_t \sqrt{1-\beta_2^t}}{1-\beta_1^t} \odot \frac{m_t}{\sqrt{v_t} + \epsilon}
\end{aligned}
$$

namely $\frac{\alpha_t \sqrt{1-\beta_2^t}}{1-\beta_1^t}$ is the actual learning rate of step t.

Implementation: TensorFlow.

```
# Copyright (C) 2020 Tong Jia. All rights reserved.
from tensorflow.python.eager import context
from tensorflow.python.framework import ops
from tensorflow.python.ops import control_flow_ops
from tensorflow.python.ops import math_ops
from tensorflow.python.ops import resource_variable_ops
from tensorflow.python.ops import state_ops
from tensorflow.python.training import optimizer
from tensorflow.python.training import training_ops


class AdamOptimizer(optimizer.Optimizer):
    """Optimizer that implements the Adam algorithm.
    References:
      Adam - A Method for Stochastic Optimization:
        [Kingma et al., 2015](https://arxiv.org/abs/1412.6980)
        ([pdf](https://arxiv.org/pdf/1412.6980.pdf))
    """

    def __init__(self,
                 learning_rate=0.001,
                 beta1=0.9,
                 beta2=0.999,
                 epsilon=1e-8,
```

```
               use_locking=False,
               name="Adam"):
      r"""Construct a new Adam optimizer.
      Initialization:
      $$m_0 := 0 \text{(Initialize initial 1st moment vector)}$$
      $$v_0 := 0 \text{(Initialize initial 2nd moment vector)}$$
      $$t := 0 \text{(Initialize timestep)}$$
      The update rule for `variable` with gradient `g` uses an optimization
      described at the end of section 2 of the paper:
      $$t := t + 1$$
      $$\text{lr}_t := \mathrm{learning_rate} *
        \sqrt{1 - \beta_2^t} / (1 - \beta_1^t)$$
      $$m_t := \beta_1 * m_{t-1} + (1 - \beta_1) * g$$
      $$v_t := \beta_2 * v_{t-1} + (1 - \beta_2) * g * g$$
      $$\text{variable} := \text{variable} -
        \text{lr}_t * m_t / (\sqrt{v_t} + \epsilon)$$
      The default value of 1e-8 for epsilon might not be a good default in
      general. For example, when training an Inception network on ImageNet a
      current good choice is 1.0 or 0.1. Note that since AdamOptimizer uses the
      formulation just before Section 2.1 of the Kingma and Ba paper rather than
      the formulation in Algorithm 1, the "epsilon" referred to here is "epsilon
      hat" in the paper.
      The sparse implementation of this algorithm (used when the gradient is an
      IndexedSlices object, typically because of `tf.gather` or an embedding
      lookup in the forward pass) does apply momentum to variable slices even if
      they were not used in the forward pass (meaning they have a gradient equal
      to zero). Momentum decay (beta1) is also applied to the entire momentum
      accumulator. This means that the sparse behavior is equivalent to the dense
      behavior (in contrast to some momentum implementations which ignore momentum
      unless a variable slice was actually used).
      Args:
        learning_rate: A Tensor or a floating point value.  The learning rate.
        beta1: A float value or a constant float tensor. The exponential decay
          rate for the 1st moment estimates.
        beta2: A float value or a constant float tensor. The exponential decay
          rate for the 2nd moment estimates.
        epsilon: A small constant for numerical stability. This epsilon is
          "epsilon hat" in the Kingma and Ba paper (in the formula just before
          Section 2.1), not the epsilon in Algorithm 1 of the paper.
        use_locking: If True use locks for update operations.
        name: Optional name for the operations created when applying gradients.
          Defaults to "Adam".  @compatibility(eager) When eager execution is
          enabled, `learning_rate`, `beta1`, `beta2`, and `epsilon` can each be a
          callable that takes no arguments and returns the actual value to use.
          This can be useful for changing these values across different
          invocations of optimizer functions. @end_compatibility
      """
      super(AdamOptimizer, self).__init__(use_locking, name)
      self._lr = learning_rate
      self._beta1 = beta1
      self._beta2 = beta2
      self._epsilon = epsilon

      # Tensor versions of the constructor arguments, created in _prepare().
      self._lr_t = None
      self._beta1_t = None
      self._beta2_t = None
      self._epsilon_t = None

  def _get_beta_accumulators(self):
      with ops.init_scope():
          if context.executing_eagerly():
              graph = None
          else:
              graph = ops.get_default_graph()
          return (self._get_non_slot_variable("beta1_power", graph=graph),
                  self._get_non_slot_variable("beta2_power", graph=graph))

  def _create_slots(self, var_list):
      # Create the beta1 and beta2 accumulators on the same device as the first
      # variable. Sort the var_list to make sure this device is consistent across
      # workers (these need to go on the same PS, otherwise some updates are
      # silently ignored).
      first_var = min(var_list, key=lambda x: x.name)
      self._create_non_slot_variable(
          initial_value=self._beta1, name="beta1_power", colocate_with=first_var)
      self._create_non_slot_variable(
          initial_value=self._beta2, name="beta2_power", colocate_with=first_var)

      # Create slots for the first and second moments.
      for v in var_list:
```

```
            self._zeros_slot(v, "m", self._name)
            self._zeros_slot(v, "v", self._name)

    def _prepare(self):
        lr = self._call_if_callable(self._lr)
        beta1 = self._call_if_callable(self._beta1)
        beta2 = self._call_if_callable(self._beta2)
        epsilon = self._call_if_callable(self._epsilon)

        self._lr_t = ops.convert_to_tensor(lr, name="learning_rate")
        self._beta1_t = ops.convert_to_tensor(beta1, name="beta1")
        self._beta2_t = ops.convert_to_tensor(beta2, name="beta2")
        self._epsilon_t = ops.convert_to_tensor(epsilon, name="epsilon")

    def _apply_dense(self, grad, var):
        m = self.get_slot(var, "m")
        v = self.get_slot(var, "v")
        beta1_power, beta2_power = self._get_beta_accumulators()
        return training_ops.apply_adam(
            var,
            m,
            v,
            math_ops.cast(beta1_power, var.dtype.base_dtype),
            math_ops.cast(beta2_power, var.dtype.base_dtype),
            math_ops.cast(self._lr_t, var.dtype.base_dtype),
            math_ops.cast(self._beta1_t, var.dtype.base_dtype),
            math_ops.cast(self._beta2_t, var.dtype.base_dtype),
            math_ops.cast(self._epsilon_t, var.dtype.base_dtype),
            grad,
            use_locking=self._use_locking).op

    def _resource_apply_dense(self, grad, var):
        m = self.get_slot(var, "m")
        v = self.get_slot(var, "v")
        beta1_power, beta2_power = self._get_beta_accumulators()
        return training_ops.resource_apply_adam(
            var.handle,
            m.handle,
            v.handle,
            math_ops.cast(beta1_power, grad.dtype.base_dtype),
            math_ops.cast(beta2_power, grad.dtype.base_dtype),
            math_ops.cast(self._lr_t, grad.dtype.base_dtype),
            math_ops.cast(self._beta1_t, grad.dtype.base_dtype),
            math_ops.cast(self._beta2_t, grad.dtype.base_dtype),
            math_ops.cast(self._epsilon_t, grad.dtype.base_dtype),
            grad,
            use_locking=self._use_locking)

    def _apply_sparse_shared(self, grad, var, indices, scatter_add):
        beta1_power, beta2_power = self._get_beta_accumulators()
        beta1_power = math_ops.cast(beta1_power, var.dtype.base_dtype)
        beta2_power = math_ops.cast(beta2_power, var.dtype.base_dtype)
        lr_t = math_ops.cast(self._lr_t, var.dtype.base_dtype)
        beta1_t = math_ops.cast(self._beta1_t, var.dtype.base_dtype)
        beta2_t = math_ops.cast(self._beta2_t, var.dtype.base_dtype)
        epsilon_t = math_ops.cast(self._epsilon_t, var.dtype.base_dtype)
        lr = (lr_t * math_ops.sqrt(1 - beta2_power) / (1 - beta1_power))
        # m_t = beta1 * m + (1 - beta1) * g_t
        m = self.get_slot(var, "m")
        m_scaled_g_values = grad * (1 - beta1_t)
        m_t = state_ops.assign(m, m * beta1_t, use_locking=self._use_locking)
        with ops.control_dependencies([m_t]):
            m_t = scatter_add(m, indices, m_scaled_g_values)
        # v_t = beta2 * v + (1 - beta2) * (g_t * g_t)
        v = self.get_slot(var, "v")
        v_scaled_g_values = (grad * grad) * (1 - beta2_t)
        v_t = state_ops.assign(v, v * beta2_t, use_locking=self._use_locking)
        with ops.control_dependencies([v_t]):
            v_t = scatter_add(v, indices, v_scaled_g_values)
        v_sqrt = math_ops.sqrt(v_t)
        var_update = state_ops.assign_sub(
            var, lr * m_t / (v_sqrt + epsilon_t), use_locking=self._use_locking)
        return control_flow_ops.group(*[var_update, m_t, v_t])

    def _apply_sparse(self, grad, var):
        return self._apply_sparse_shared(
            grad.values,
            var,
            grad.indices,
            lambda x, i, v: state_ops.scatter_add(  # pylint: disable=g-long-lambda
                x,
```

```
                i,
                v,
                use_locking=self._use_locking))

    def _resource_scatter_add(self, x, i, v):
        with ops.control_dependencies(
                [resource_variable_ops.resource_scatter_add(x.handle, i, v)]):
            return x.value()

    def _resource_apply_sparse(self, grad, var, indices):
        return self._apply_sparse_shared(grad, var, indices,
                                         self._resource_scatter_add)

    def _finish(self, update_ops, name_scope):
        # Update the power accumulators.
        with ops.control_dependencies(update_ops):
            beta1_power, beta2_power = self._get_beta_accumulators()
            with ops.colocate_with(beta1_power):
                update_beta1 = beta1_power.assign(
                    beta1_power * self._beta1_t, use_locking=self._use_locking)
                update_beta2 = beta2_power.assign(
                    beta2_power * self._beta2_t, use_locking=self._use_locking)
        return control_flow_ops.group(
            *update_ops + [update_beta1, update_beta2], name=name_scope)
```

Implementation: PyTorch.

```
# Copyright (C) 2020 Tong Jia. All rights reserved.
import math
import torch
from torch.optim.optimizer import Optimizer


class Adam(Optimizer):
    r"""Implements Adam algorithm.
    It has been proposed in `Adam: A Method for Stochastic Optimization`_.
    Arguments:
        params (iterable): iterable of parameters to optimize or dicts defining
            parameter groups
        lr (float, optional): learning rate (default: 1e-3)
        betas (Tuple[float, float], optional): coefficients used for computing
            running averages of gradient and its square (default: (0.9, 0.999))
        eps (float, optional): term added to the denominator to improve
            numerical stability (default: 1e-8)
        weight_decay (float, optional): weight decay (L2 penalty) (default: 0)
        amsgrad (boolean, optional): whether to use the AMSGrad variant of this
            algorithm from the paper `On the Convergence of Adam and Beyond`_
            (default: False)
    .. _Adam\: A Method for Stochastic Optimization:
        https://arxiv.org/abs/1412.6980
    .. _On the Convergence of Adam and Beyond:
        https://openreview.net/forum?id=ryQu7f-RZ
    """

    def __init__(self, params, lr=1e-3, betas=(0.9, 0.999), eps=1e-8,
                 weight_decay=0, amsgrad=False):
        if not 0.0 <= lr:
            raise ValueError("Invalid learning rate: {}".format(lr))
        if not 0.0 <= eps:
            raise ValueError("Invalid epsilon value: {}".format(eps))
        if not 0.0 <= betas[0] < 1.0:
            raise ValueError("Invalid beta parameter at index 0: {}".format(betas[0]))
        if not 0.0 <= betas[1] < 1.0:
            raise ValueError("Invalid beta parameter at index 1: {}".format(betas[1]))
        defaults = dict(lr=lr, betas=betas, eps=eps,
                        weight_decay=weight_decay, amsgrad=amsgrad)
        super(Adam, self).__init__(params, defaults)

    def __setstate__(self, state):
        super(Adam, self).__setstate__(state)
        for group in self.param_groups:
            group.setdefault('amsgrad', False)

    def step(self, closure=None):
        """Performs a single optimization step.
        Arguments:
            closure (callable, optional): A closure that reevaluates the model
                and returns the loss.
```

```
        """
        loss = None
        if closure is not None:
            loss = closure()

        for group in self.param_groups:
            for p in group['params']:
                if p.grad is None:
                    continue
                grad = p.grad.data
                if grad.is_sparse:
                    raise RuntimeError('Adam does not support sparse gradients, please consider SparseAdam
                                                                                    instead')
                amsgrad = group['amsgrad']

                state = self.state[p]

                # State initialization
                if len(state) == 0:
                    state['step'] = 0
                    # Exponential moving average of gradient values
                    state['exp_avg'] = torch.zeros_like(p.data, memory_format=torch.preserve_format)
                    # Exponential moving average of squared gradient values
                    state['exp_avg_sq'] = torch.zeros_like(p.data, memory_format=torch.preserve_format)
                    if amsgrad:
                        # Maintains max of all exp. moving avg. of sq. grad. values
                        state['max_exp_avg_sq'] = torch.zeros_like(p.data, memory_format=torch.
                                                                        preserve_format)

                exp_avg, exp_avg_sq = state['exp_avg'], state['exp_avg_sq']
                if amsgrad:
                    max_exp_avg_sq = state['max_exp_avg_sq']
                beta1, beta2 = group['betas']

                state['step'] += 1
                bias_correction1 = 1 - beta1 ** state['step']
                bias_correction2 = 1 - beta2 ** state['step']

                if group['weight_decay'] != 0:
                    grad = grad.add(group['weight_decay'], p.data)

                # Decay the first and second moment running average coefficient
                exp_avg.mul_(beta1).add_(1 - beta1, grad)
                exp_avg_sq.mul_(beta2).addcmul_(1 - beta2, grad, grad)
                if amsgrad:
                    # Maintains the maximum of all 2nd moment running avg. till now
                    torch.max(max_exp_avg_sq, exp_avg_sq, out=max_exp_avg_sq)
                    # Use the max. for normalizing running avg. of gradient
                    denom = (max_exp_avg_sq.sqrt() / math.sqrt(bias_correction2)).add_(group['eps'])
                else:
                    denom = (exp_avg_sq.sqrt() / math.sqrt(bias_correction2)).add_(group['eps'])

                step_size = group['lr'] / bias_correction1

                p.data.addcdiv_(-step_size, exp_avg, denom)

        return loss
```

4.1.8 AdaMax

Algorithm 8: AdaMax algorithm, a variant of Adam based on the infinity norm, for stochastic optimization[29].

Input: α: Stepsize;
$\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates;
$\epsilon \in \mathbb{R}$: Small constant, perhaps 10^{-8} for numerical stability;
$m_0 \in \mathbb{R}^n$: Initial moving 1st moment, perhaps 0 at very beginning;
$v_0 \in \mathbb{R}^n$: Initial moving 2nd moment, perhaps 0 at very beginning;
$\theta_0 \in \mathbb{R}^n$: Initial parameters vector.

Output: $\theta_T \in \mathbb{R}^n$: Resulting parameters.

for $t \in \{1, \ldots, T\}$ **do**
  $\{(x^{(i)}, y^{(i)})\}_{i=1}^{M} \sim \mathcal{D}$ ▷ Sample a minibatch of M samples with corresponding targets from training set $\mathcal{D}$
  $g_t \leftarrow \frac{1}{M} \sum_{i=1}^{M} \nabla_\theta \mathcal{L}\left(f(x^{(i)}; \theta_{t-1}), y^{(i)}\right)$ ▷ Calculate gradients w.r.t. stochastic objective at timestep t
  $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$ ▷ Update biased 1st moment estimate
  $v_t \leftarrow \max(\beta_2 \odot v_{t-1}, |g_t|)$ ▷ Update the exponentially weighted infinity norm
  $\theta_t \leftarrow \theta_{t-1} - \frac{\alpha}{1-\beta_1^t} \odot \frac{m_t}{v_t}$ ▷ Update parameters
end
return θ_T ▷ Resulting parameters

4.1.9 NAdam

4.1.10 AMSGrad

4.1.11 AdaBound

4.1.12 RAdam

4.1.13 Lookahead

4.1.14 AdaMod

Algorithm 9: AdaMod algorithm for stochastic optimization[30].

Input: $\{\alpha_t \in \mathbb{R}\}_{t=1}^{T}$: Stepsize of each iteration;
$\beta_1, \beta_2, \beta_3 \in [0, 1)$: Moment decay;
$\epsilon \in \mathbb{R}$: Small constant, perhaps 10^{-8} for numerical stability;
$m_0 \in \mathbb{R}^n$: Initial moving 1st moment, perhaps 0 at very beginning;
$v_0 \in \mathbb{R}^n$: Initial moving 2nd moment, perhaps 0 at very beginning;
$s_0 \in \mathbb{R}^n$: Initial smoothed element-wise step size, perhaps 0 at very beginning;
$\theta_0 \in \mathbb{R}^n$: Initial parameters.

Output: $\theta_T \in \mathbb{R}^n$: Resulting parameters.

for $t \in \{1, \ldots, T\}$ **do**
  $\{(x^{(i)}, y^{(i)})\}_{i=1}^{M} \sim \mathcal{D}$ ▷ Sample a minibatch of M samples with corresponding targets from training set $\mathcal{D}$
  $g_t \leftarrow \frac{1}{M} \sum_{i=1}^{M} \nabla_\theta \mathcal{L}\left(f(x^{(i)}; \theta_{t-1}), y^{(i)}\right)$ ▷ Calculate gradients w.r.t. stochastic objective at timestep t
  $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$ ▷ Update biased 1st moment estimate
  $v_t \leftarrow \beta v_{t-1} + (1 - \beta) g_t \odot g_t$ ▷ Update biased 2nd moment estimate
  $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ ▷ Compute bias-corrected 1st moment estimate
  $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ ▷ Compute bias-corrected 2nd moment estimate
  $\eta_t \leftarrow \alpha_t / (\sqrt{\widehat{v}_t} + \epsilon)$ ▷ Compute element-wise adaptive stepsizes
  $s_t \leftarrow \beta_3 s_{t-1} + (1 - \beta_3) \eta_t$ ▷ Smoothing adaptive stepsizes
  $\widehat{\eta}_t \leftarrow \min(\eta_t, s_t)$ ▷ Bounding adaptive stepsizes
  $\theta_t \leftarrow \theta_{t-1} - \widehat{\eta}_t \odot \widehat{m}_t$ ▷ Update parameters
end
return θ_T ▷ Resulting parameters

4.2 Second-order Gradient Optimization Methods

4.2.1 K-FAC

Reading Materials.

- Optimizing Neural Networks with Kronecker-factored Approximate Curvature (Martens & Grosse, 2015)[31].

4.2.2 Shampoo

Algorithm 10: Shampoo for matrix case[32].

Input: $\{\alpha_t \in \mathbb{R}\}_{t=1}^{T}$: Stepsize of each iteration;
$\beta_1, \beta_2 \in (0, 1]$: Moment decay;
$W^{(0)} \in \mathbb{R}^{m \times n}$: Initial parameters form a matrix.

Output: $W^{(T)} \in \mathbb{R}^n$: Resulting parameters.

for $t \in \{1, \ldots, T\}$ **do**
  $\mathcal{D}_{\mathcal{B}} = \{(x, y)\} \sim \mathcal{D}$ ▷ Sample a minibatch of samples with corresponding targets from training set $\mathcal{D}$
  $G^{(t)} \leftarrow \frac{1}{|\mathcal{D}_{\mathcal{B}}^{(t)}|} \sum_{(x,y) \in \mathcal{D}_{\mathcal{B}}^{(t)}} \nabla_W \mathcal{L}(f_\theta(x), y)$ ▷ Calculate gradients w.r.t. stochastic objective at timestep t
  $L^{(t)} \leftarrow L^{(t-1)} + G^{(t)}(G^{(t)})^{\mathrm{T}}$
  $R^{(t)} \leftarrow R^{(t-1)} + (G^{(t)})^{\mathrm{T}} G^{(t)}$ ▷ Get preconditioner statistics computation of t-th iterational update
  $W^{(t+1)} \leftarrow W^{(t)} - \widetilde{\alpha}_t \odot \left({L^{(t)}}^{-\frac{1}{4}} G^{(t)} {R^{(t)}}^{-\frac{1}{4}} \right)$ ▷ Update parameters, need to compute inverse $1/4$-th root ${L^{(t)}}^{-1/4}$ and ${R^{(t)}}^{-1/4}$ first
end
return θ_T ▷ Resulting parameters

Shampoo for Matrices. In the two dimensional case, the trainable parameters form a matrix $W \in \mathbb{R}^{m \times n}$ (e.g. linear parameters for softmax regression). First-order methods update iterates $W^{(t)} \in \mathbb{R}^{m \times n}$ based on the gradient $G^{(t)} = \frac{1}{|\mathcal{D}_{\mathcal{B}}^{(t)}|} \sum_{(x,y) \in \mathcal{D}_{\mathcal{B}}^{(t)}} \nabla_W \mathcal{L}(f_\Theta(x), y) \in \mathbb{R}^{m \times n}$.

$$W^{(t)} = \begin{pmatrix} W_{11}^{(t)} & W_{12}^{(t)} & \cdots & W_{1n}^{(t)} \\ W_{21}^{(t)} & W_{22}^{(t)} & \cdots & W_{2n}^{(t)} \\ \vdots & \vdots & \vdots & \vdots \\ W_{m1}^{(t)} & W_{m2}^{(t)} & \cdots & W_{mn}^{(t)} \end{pmatrix} \in \mathbb{R}^{m \times n}$$

$$G^{(t)} = \frac{1}{|\mathcal{D}_{\mathcal{B}}^{(t)}|} \sum_{(x,y) \in \mathcal{D}_{\mathcal{B}}^{(t)}} \begin{pmatrix} \frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{11}} & \frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{12}} & \cdots & \frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{1n}} \\ \frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{21}} & \frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{22}} & \cdots & \frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{2n}} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{m1}} & \frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{m2}} & \cdots & \frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{mn}} \end{pmatrix} \in \mathbb{R}^{m \times n}$$

where $|\mathcal{D}_{\mathcal{B}}^{(t)}|$ means the number samples in mini-batch dataset of t-th iteration $\mathcal{D}_{\mathcal{B}}^{(t)} \sim \mathcal{D}$. The Shampoo algorithm tracks two statistics $L^{(t)}$ and $R^{(t)}$ over the iteration of its run, which are dened as follows:

$$L^{(t)} = \sum_{\ell=1}^{t} G^{(\ell)}(G^{(\ell)})^{\mathrm{T}} \in \mathbb{R}^{m \times m}; \quad R^{(t)} = \sum_{\ell=1}^{t} (G^{(\ell)})^{\mathrm{T}} G^{(\ell)} \in \mathbb{R}^{n \times n}$$

where in ℓ-th iteration, we have $G^{(\ell)}(G^{(\ell)})^{\mathrm{T}}$ in $L^{(\ell)}$ as follows:

$$\begin{aligned} & G^{(\ell)}(G^{(\ell)})^{\mathrm{T}} \\ =& \frac{1}{|\mathcal{D}_{\mathcal{B}}^{(\ell)}|} \sum_{(x,y) \in \mathcal{D}_{\mathcal{B}}^{(\ell)}} \left(\begin{pmatrix} \frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{11}} & \frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{12}} & \cdots & \frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{1n}} \\ \frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{21}} & \frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{22}} & \cdots & \frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{2n}} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{m1}} & \frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{m2}} & \cdots & \frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{mn}} \end{pmatrix} \begin{pmatrix} \frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{11}} & \frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{21}} & \cdots & \frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{m1}} \\ \frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{12}} & \frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{22}} & \cdots & \frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{m2}} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{1n}} & \frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{2n}} & \cdots & \frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{mn}} \end{pmatrix} \right) \\ =& \frac{1}{|\mathcal{D}_{\mathcal{B}}^{(\ell)}|} \sum_{(x,y) \in \mathcal{D}_{\mathcal{B}}^{(\ell)}} \begin{pmatrix} \sum_{j=1}^{n} \left(\frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{1j}} \frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{1j}} \right) & \sum_{j=1}^{n} \left(\frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{1j}} \frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{2j}} \right) & \cdots & \sum_{j=1}^{n} \left(\frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{1j}} \frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{mj}} \right) \\ \sum_{j=1}^{n} \left(\frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{2j}} \frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{1j}} \right) & \sum_{j=1}^{n} \left(\frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{2j}} \frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{2j}} \right) & \cdots & \sum_{j=1}^{n} \left(\frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{2j}} \frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{mj}} \right) \\ \vdots & \vdots & \vdots & \vdots \\ \sum_{j=1}^{n} \left(\frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{mj}} \frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{1j}} \right) & \sum_{j=1}^{n} \left(\frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{mj}} \frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{2j}} \right) & \cdots & \sum_{j=1}^{n} \left(\frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{mj}} \frac{\partial \mathcal{L}(f_\Theta(x),y)}{\partial W_{mj}} \right) \end{pmatrix} \end{aligned}$$

$$=\frac{1}{|\mathcal{D}_{\mathcal{B}}^{(\ell)}|}\sum_{(x,y)\in\mathcal{D}_{\mathcal{B}}^{(\ell)}}\begin{pmatrix}\sum_{j=1}^{n}\left(\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{1j}}\right)^2 & \sum_{j=1}^{n}\left(\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{1j}}\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{2j}}\right) & \cdots & \sum_{j=1}^{n}\left(\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{1j}}\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{mj}}\right)\\ \sum_{j=1}^{n}\left(\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{2j}}\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{1j}}\right) & \sum_{j=1}^{n}\left(\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{2j}}\right)^2 & \cdots & \sum_{j=1}^{n}\left(\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{2j}}\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{mj}}\right)\\ \vdots & \vdots & \vdots & \vdots\\ \sum_{j=1}^{n}\left(\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{mj}}\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{1j}}\right) & \sum_{j=1}^{n}\left(\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{mj}}\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{2j}}\right) & \cdots & \sum_{j=1}^{n}\left(\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{mj}}\right)^2\end{pmatrix}$$

and we have $(G^{(\ell)})^{\mathrm{T}}G^{(\ell)}$ in $R^{(\ell)}$ as follows:

$$(G^{(\ell)})^{\mathrm{T}}G^{(\ell)}$$

$$=\frac{1}{|\mathcal{D}_{\mathcal{B}}^{(\ell)}|}\sum_{(x,y)\in\mathcal{D}_{\mathcal{B}}^{(\ell)}}\left(\begin{pmatrix}\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{11}} & \frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{21}} & \cdots & \frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{m1}}\\ \frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{12}} & \frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{22}} & \cdots & \frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{m2}}\\ \vdots & \vdots & \vdots & \vdots\\ \frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{1n}} & \frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{2n}} & \cdots & \frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{mn}}\end{pmatrix}\begin{pmatrix}\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{11}} & \frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{12}} & \cdots & \frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{1n}}\\ \frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{21}} & \frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{22}} & \cdots & \frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{2n}}\\ \vdots & \vdots & \vdots & \vdots\\ \frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{m1}} & \frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{m2}} & \cdots & \frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{mn}}\end{pmatrix}\right)$$

$$=\frac{1}{|\mathcal{D}_{\mathcal{B}}^{(\ell)}|}\sum_{(x,y)\in\mathcal{D}_{\mathcal{B}}^{(\ell)}}\begin{pmatrix}\sum_{i=1}^{m}\left(\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{i1}}\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{i1}}\right) & \sum_{i=1}^{m}\left(\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{i1}}\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{i2}}\right) & \cdots & \sum_{i=1}^{m}\left(\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{i1}}\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{in}}\right)\\ \sum_{i=1}^{m}\left(\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{i2}}\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{i1}}\right) & \sum_{i=1}^{m}\left(\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{i2}}\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{i2}}\right) & \cdots & \sum_{i=1}^{m}\left(\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{i2}}\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{in}}\right)\\ \vdots & \vdots & \vdots & \vdots\\ \sum_{i=1}^{m}\left(\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{in}}\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{i1}}\right) & \sum_{i=1}^{m}\left(\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{in}}\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{i2}}\right) & \cdots & \sum_{i=1}^{m}\left(\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{in}}\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{in}}\right)\end{pmatrix}$$

$$=\frac{1}{|\mathcal{D}_{\mathcal{B}}^{(\ell)}|}\sum_{(x,y)\in\mathcal{D}_{\mathcal{B}}^{(\ell)}}\begin{pmatrix}\sum_{i=1}^{m}\left(\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{i1}}\right)^2 & \sum_{i=1}^{m}\left(\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{i1}}\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{i2}}\right) & \cdots & \sum_{i=1}^{m}\left(\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{i1}}\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{in}}\right)\\ \sum_{i=1}^{m}\left(\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{i2}}\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{i1}}\right) & \sum_{i=1}^{m}\left(\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{i2}}\right)^2 & \cdots & \sum_{i=1}^{m}\left(\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{i2}}\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{in}}\right)\\ \vdots & \vdots & \vdots & \vdots\\ \sum_{i=1}^{m}\left(\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{in}}\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{i1}}\right) & \sum_{i=1}^{m}\left(\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{in}}\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{i2}}\right) & \cdots & \sum_{i=1}^{m}\left(\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{in}}\right)^2\end{pmatrix}$$

where $\sum$ represents matrix sum operation and note that $\left(\frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{ij}}\right)^2 \neq \frac{\partial\mathcal{L}(f_\Theta(x),y)}{\partial W_{ij}^2}$. Both symmetric matrices $L^{(t)} \in \mathbb{R}^{m\times m}$ and $R^{(t)} \in \mathbb{R}^{n\times n}$ are used to precondition gradient and update W as follows:

$$W^{(t+1)} := W^{(t)} - \alpha_t {L^{(t)}}^{-\frac{1}{4}} G^{(t)} {R^{(t)}}^{-\frac{1}{4}}$$

where $\alpha_t > 0$ represents the stepsize of t-th iteration. The primary complexity of Shampoo arises from computing ${L^{(t)}}^{-\frac{1}{4}}$ and ${R^{(t)}}^{-\frac{1}{4}}$ which was computed using singular value decomposition which is expensive.

Shampoo for Tensors.

Reading Materials.

- Shampoo: Preconditioned Stochastic Tensor Optimization (Gupta et al., 2018)[32].
- Second Order Optimization Made Practical (Anil et al., 2020)[33].

4.3 Additional Strategies for Optimizing SGD

4.3.1 Decoupled Weight Decay (Not Accomplished)

Intuition. Consider the objective with L_2 regularization:

$$\mathcal{J}(\theta) = \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}\left(f_\theta(x^{(i)}), y^{(i)}\right) + \frac{\lambda}{2}\|\theta\|_2^2$$

if we use standard SGD optimization algorithm, at each training iteration t, we can get the gradients with respect to stochastic objective:

$$g_t \leftarrow \frac{1}{M}\sum_{i=1}^{M}\nabla_\theta\mathcal{L}\left(f(x^{(i)}; \theta_{t-1}), y^{(i)}\right) + \lambda\theta_{t-1}$$

Proposition 4.1 (Weight decay $=$ L_2 reg for standard SGD)**.**

Proposition 4.2 (Weight decay $\neq$ L_2 reg for adaptive gradients)**.**

Proposition 4.3 (Weight decay $=$ scale-adjusted L_2 reg for adaptive gradient algorithm with fixed preconditioner)**.**

Definition. Pseudocode of SGDW and AdamW show following:

Algorithm 11: SGD with L_2 regularization and SGD with decoupled weight decay (SGDW), both with momentum for stochastic optimization[34].

Input: $\alpha \in \mathbb{R}$: Initial stepsize;
$\beta \in \mathbb{R}$: Momentum factor;
$\lambda \in \mathbb{R}$: Weight decay/L_2 regularization factor;
$m_0 \in \mathbb{R}^n$: Initial moving 1st moment, perhaps 0 at very beginning;
$\eta_0 \in \mathbb{R}$: Initial schedule multiplier;
$\theta_0 \in \mathbb{R}^n$: Initial parameters.

Output: $\theta_T \in \mathbb{R}^n$: Resulting parameters.

for $t \in \{1, \ldots, T\}$ **do**
  $\{(x^{(i)}, y^{(i)})\}_{i=1}^{M} \sim \mathcal{D}$ ▷ Sample a minibatch of M samples with corresponding targets from training set $\mathcal{D}$
  $g_t \leftarrow \frac{1}{M} \sum_{i=1}^{M} \nabla_\theta \mathcal{L}\left(f(x^{(i)}; \theta_{t-1}), y^{(i)}\right) + \lambda\theta_{t-1}$ ▷ Calculate gradients w.r.t. stochastic obj at timestep t
  $\eta_t \leftarrow \text{SetScheduleMultiplier}(t)$ ▷ Introduce a scaling factor, can be fixed, decay, be used for warm restarts
  $m_t \leftarrow \beta m_{t-1} + \eta_t \alpha g_t$ ▷ Update exponential moving 1st moment
  $\theta_t \leftarrow \theta_{t-1} - m_t - \eta_t \lambda \theta_{t-1}$ ▷ Update parameters
end
return θ_T

Algorithm 12: Adam with L_2 regularization and Adam with decoupled weight decay (AdamW) for stochastic optimization[34].

Input: $\alpha \in \mathbb{R}$: Initial stepsize;
$\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates to calculate 1st and 2nd moment, perhaps $(0.9, 0.999)$;
$\epsilon \in \mathbb{R}$: Small constant, perhaps 10^{-8} for numerical stability;
$m_0 \in \mathbb{R}^n$: Initial moving 1st moment, perhaps 0 at very beginning;
$v_0 \in \mathbb{R}^n$: Initial moving 2nd moment, perhaps 0 at very beginning;
$\eta_0 \in \mathbb{R}$: Initial schedule multiplier;
$\theta_0 \in \mathbb{R}^n$: Initial parameters.

Output: $\theta_T \in \mathbb{R}^n$: Resulting parameters.

for $t \in \{1, \ldots, T\}$ **do**
  $\{(x^{(i)}, y^{(i)})\}_{i=1}^{M} \sim \mathcal{D}$ ▷ Sample a minibatch of M samples with corresponding targets from training set $\mathcal{D}$
  $g_t \leftarrow \frac{1}{M} \sum_{i=1}^{M} \nabla_\theta \mathcal{L}\left(f(x^{(i)}; \theta_{t-1}), y^{(i)}\right) + \lambda\theta_{t-1}$ ▷ Calculate gradients w.r.t. stochastic obj at timestep t
  $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$ ▷ Update biased 1st moment estimate
  $v_t \leftarrow \beta v_{t-1} + (1 - \beta) g_t \odot g_t$ ▷ Update biased 2nd moment estimate
  $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ ▷ Compute bias-corrected 1st moment estimate
  $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ ▷ Compute bias-corrected 2nd moment estimate
  $\eta_t \leftarrow \text{SetScheduleMultiplier}(t)$ ▷ Introduce a scaling factor, can be fixed, decay, be used for warm restarts
  $\theta_t \leftarrow \theta_{t-1} - \eta_t (\alpha \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon) + \lambda \theta_{t-1})$ ▷ Update parameters
end
return θ_T

4.3.2 Gradient Noise

Methodology. According to paper: Adding Gradient Noise Improves Learning for Very Deep Networks[35], consider a simple technique of adding time-dependent Gaussian noise to the gradient g_t at every training step t:

$$g_t \leftarrow g_t + \mathcal{N}(0, \sigma_t^2)$$

their experiments indicate that adding annealed Gaussian noise by decaying the variance works better than using fixed Gaussian noise, they use a schedule of variance that:

$$\sigma_t^2 = \frac{\eta}{(1+t)^\gamma}$$

where η (**not stepsize!**) selected from $\{0.01, 0.3, 1.0\}$ and $\gamma = 0.55$. Higher gradient noise at the beginning of training forces the gradient away from 0 in the early stages.

4.3.3 Switching Optimizers

Algorithm 13: Improving Generalization Performance by Switching from Adam to SGD[36].

Input: α: Initial stepsize;
β_1, β_2

Part II
Machine Learning Algorithms

5 Logistic Regression

5.1 Vanilla Logistic Regression

5.1.1 Notation

All notations used in this Subsection 5.1 in the derivation process of LR are shown as follows:

5.1.2 Definition

Inference Logistic regression (LR) model is a linear model for binary classification problems. The probability prediction function can be expressed as

$$\Pr(y=1|\boldsymbol{x}) = f(\boldsymbol{x};\boldsymbol{w}) = \frac{1}{1+\exp(-\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x})}$$

where $\boldsymbol{x} \in \mathbb{R}^d$ represents a feature vector and $\boldsymbol{w} \in \mathbb{R}^d$ means the linear trainable parameters.

The LR function $f(\boldsymbol{x};\boldsymbol{w})$ consists two parts:

- Internal function: $z = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} = \sum_{j=1}^{d} w_j x_j \in \mathbb{R}$, a linear function.
- External function: $\sigma(z) = \frac{1}{1+\exp(-z)} \in (0,1)$, a sigmoid function.

Combine the internal linear function and external sigmoid function, we can get the probability inference formulation of logistic regression:

$$\begin{aligned} f(\boldsymbol{x};\boldsymbol{w}) &= \sigma(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}) \\ &= \frac{1}{1+\exp(-\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x})} \\ &= \frac{1}{1+\exp(-\sum_{j=1}^{d} w_j x_j)} \in (0,1) \end{aligned}$$

and the corresponding discriminant prediction function can be expressed as follows:

$$h(\boldsymbol{x}; f_{\boldsymbol{w}}(\cdot); \alpha) = \begin{cases} 0 & \text{if } f_{\boldsymbol{w}}(\boldsymbol{x}) \leq \alpha \\ 1 & \text{if } f_{\boldsymbol{w}}(\boldsymbol{x}) > \alpha \end{cases}$$

where threshold $\alpha \in (0,1)$ (e.g., $\alpha = 0.5$) is a hyperparameter.

Trainable Parameters

$$\Theta = \{\boldsymbol{w} \in \mathbb{R}^d\}$$

where $\mathrm{w}_j \in \mathbb{R}$ represents the weight corresponging the j-th dimension of the feature vector x_j.

5.1.3 Learning

We will make derivations for back-propagation of logistic regression model (i.e., learning process) in ***sample-level***.

Training Dataset

$$\mathcal{D} = \{(x^{(i)}, y^{(i)}) | x^{(i)} \in \mathbb{R}^d, y^{(i)} \in \{0,1\}\}_{i=1}^{N}$$

where each training explanatory vector (i.e., feature vector) is a d-dimensional vector and corresponding response variable (i.e., label) takes binary value. $y^{(i)} = 1$ means predicting the corresponding sample $x^{(i)}$ as a positive class with a 100% probability.

Objective We start the derivation from the most primitive maximum Llikelihood estimation 3.2.2 (MLE):

$$\begin{aligned} \max_{\Theta} \Pr(\mathcal{D};\Theta) &= \Pr\left(\{(x^{(i)}, y^{(i)})\}_{i=1}^{N};\Theta\right) \\ &= \prod_{i=1}^{N} \Pr(x^{(i)}, y^{(i)};\Theta) \quad (\because \text{i.i.d.}) \end{aligned}$$

$$
\begin{aligned}
&= \prod_{i=1}^{N} \Pr(y^{(i)}|x^{(i)};\Theta) \Pr(x^{(i)}) \\
&= \prod_{i=1}^{N} \Pr(y^{(i)}|x^{(i)};\Theta) \prod_{i=1}^{N} \Pr(x^{(i)})
\end{aligned}
$$

For the sake of neatness, we did not use bold notations during the derivation process. Apply a logarithmic operation $\log(\cdot)$ to the original probability $\Pr(\mathcal{D};\Theta)$, besides considering the probability density of each observation $(x^{(i)}, y^{(i)})$, namely $\frac{1}{N}$, we have an equivalent conversion of likelihood objective without changing monotony as follows:

$$
\max_{\Theta} \ \mathcal{J}(\Theta) = \frac{1}{N} \sum_{i=1}^{N} \log \Pr(y^{(i)}|x^{(i)};\Theta)
$$

Convert above maximizing likelihood objective equivalently to minimize KL divergence, we can continue to do the following derivation:

$$
\begin{aligned}
\min_{\Theta} \ \mathcal{L}(\Theta) &= \frac{1}{N} \sum_{i=1}^{N} \mathrm{KL}(p^{(i)} \| q^{(i)}) \quad (y^{(i)} \in \{1, \cdots, K\} \sim p^{(i)} \in \mathbb{P}^K, q^{(i)} = q_{\Theta}^{(i)} \in \mathbb{P}^K) \\
&= \frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{K} p_k^{(i)} \log \frac{p_k^{(i)}}{q_k^{(i)}} \quad (\because \mathrm{KL}(p^{(i)} \| q^{(i)}) = \sum_{k=1}^{K} p_k^{(i)} \log \frac{p_k^{(i)}}{q_k^{(i)}})
\end{aligned}
$$

6 Softmax Regression

6.1 Vanilla Softmax Regression

6.1.1 Definition

6.1.2 Learning

6.2 Variants of Softmax Regression

6.2.1 Definition

6.2.2 Learning

6.3 Mixtape

6.3.1 Definition

Based on paper Mixtape: Breaking the Softmax Bottleneck Efficiently[37]

6.3.2 Learning

7 Naive Bayes

7.1 Naive Bayes for Classification

7.1.1 Definition

7.1.2 Learning

7.2 Naive Bayes for Regression

7.2.1 Definition

7.2.2 Learning

8 Support Vector Machines (SVMs)

8.1 Hard-Margin SVMs for Classification

8.1.1 Definition

8.1.2 Learning

8.2 Soft-Margin SVMs for Classification

8.2.1 Definition

8.2.2 Learning

8.2.3 Kernel Trick for Nonlinear Classification

9 Decision Trees

9.1 Classification & Regression Tree (CART)

9.1.1 Definition

10 Dimensionality Reduction

10.1 Principal Component Analysis (PCA)

10.1.1 Definition

10.1.2 Learning

Part III
Ensemble Learning Algorithms

11 Random Forest

12 AdaBoost

12.1 AdaBoost for Binary Classification

12.2 AdaBoost for Multi Classification

12.3 AdaBoost for Regression

13 Gradient Boosting Machine (GBM)

13.1 Vanilla Gradient Boosting Machine

13.1.1 Definition

$$\mathcal{F}(x;\{\beta_t,\theta_t\}_{t=1}^T)=\sum_{t=1}^T\beta_t f(x;\theta_t)$$

where T is the total number of weak predictive function, β_t is the weight of t-th weak predictive function and θ_t is learnable parameters of t-th weak predictive function. $\mathcal{F}:\mathcal{X}\to\mathcal{Y}$ is the final estimated strong predictive function. Note for classification task, $\mathcal{F}:\mathcal{X}\to\mathbb{R}^K$ is the logit predictive function, not probability predictive function.

13.1.2 Learning

Algorithm 14: Greedy Function Approximation: A Gradient Boosting Machine[38].

Input: $\mathcal{D}=\{(x^{(i)},y^{(i)})\}_{i=1}^N$: Training dataset;
T: Number of iterations;
$\mathcal{L}(\cdot,\cdot)$: A differentiable loss function;
$f(x;\theta_t),\forall t\in\{1,\dots,T\}$: Base estimated function with initialized parameters θ_t of each iteration;
$\beta_t=\{\beta_{t,1},\beta_{t,2},\dots,\beta_{t,k_t}\},\ \forall t\in\{1,\dots,T\}$: Candidates of multiplier of each iteration.

Output: $\mathcal{F}_T(x)=\mathcal{F}(x;\{\beta_t,\theta_t\}_{t=1}^T)$: The final estimated function.

$\mathcal{F}_0(x)=\arg\min_\alpha\sum_{i=1}^N\mathcal{L}(\alpha,y^{(i)})$ ▷ Initialize strong estimated function with a constant value

for $t=\{1,\dots,T\}$ **do**

  $r_t^{(i)}=-\left[\frac{\partial\mathcal{L}(\mathcal{F}_{t-1}(x^{(i)}),y^{(i)})}{\partial\mathcal{F}(x^{(i)})}\right]_{\mathcal{F}(x)=\mathcal{F}_{t-1}(x)}\quad\forall i\in\{1,\dots,N\}$ ▷ Compute so-called pseudo-residuals for all training samples

  $\theta_t=\arg\min_\theta\frac{1}{N}\sum_{i=1}^N\mathcal{L}(f(x^{(i)};\theta),r_t^{(i)})$ ▷ Fit $f(x;\theta_t)$ with pseudo-residuals dataset $\mathcal{D}_t=\{(x^{(i)},r_t^{(i)})\}_{i=1}^N$

  $\beta_t=\arg\min_\beta\frac{1}{N}\sum_{i=1}^N\mathcal{L}(\mathcal{F}_{t-1}(x^{(i)})+\beta f(x^{(i)};\theta_t),y^{(i)})$ ▷ Compute multiplier β_t by solving one-dimensional optimization problem

  $\mathcal{F}_t(x):=\mathcal{F}_{t-1}(x)+\beta_t f(x;\theta_t)$ ▷ Update the strong estimated function

end

return $\mathcal{F}_t(x)$

Specific so-called pseudo-residuals.

- Squared loss (regression) as loss function $\mathcal{L}$: (Fitting scalar residual)

$$\mathcal{L}(\mathcal{F}(x),y)=\frac{1}{2}\left(\mathcal{F}(x)-y\right)^2\quad(\mathcal{F}:\mathcal{X}\to\mathbb{R},y\in\mathbb{R})$$

  therefore we can get gredient as follow:

$$\frac{\partial\mathcal{L}(\mathcal{F}(x),y)}{\partial\mathcal{F}(x)}=\mathcal{F}(x)-y$$

  and the so-called pseudo-residual of a specific sample (x,y) is:

$$r=-\frac{\partial\mathcal{L}(\mathcal{F}(x),y)}{\partial\mathcal{F}(x)}=y-\mathcal{F}(x)$$

- Cross entropy (classification) as loss function $\mathcal{L}$: (Fitting probability distribution residual)

$$\mathcal{L}(\mathcal{Q}(x),y)=-y\cdot\log\mathcal{Q}(x)=-\sum_{k=1}^K y_k\log\left[\mathcal{Q}(x)\right]_k\quad(\mathcal{Q}:\mathcal{X}\to\mathbb{P}^K,\ y\in\mathbb{P}^K)$$

and we have the relation between probability predictive function $\mathcal{Q}$ and logit predictive function $\mathcal{F}$ as follow:

$$[\mathcal{Q}(x)]_k = \frac{\exp\left([\mathcal{F}(x)]_k\right)}{\sum_{l=1}^{K}\exp\left([\mathcal{F}(x)]_l\right)} \quad (\mathcal{F}: \mathcal{X} \to \mathbb{R}^K \text{(logit function!)})$$

equivalently we use the symmetric multiple logistic transform:

$$[\mathcal{F}(x)]_k = [\mathcal{Q}(x)]_k - \frac{1}{K}\sum_{l=1}^{K}[\mathcal{Q}(x)]_l$$

therefore we can derivate so-called pseudo-residual as follow:

$$r_k = -\frac{\partial \mathcal{L}(\mathcal{Q}(x), y)}{\partial\left[\mathcal{F}(x)\right]_k} = y_k - [\mathcal{Q}_{t-1}(x)]_k$$

Problem: How to proof the derivation of so-called pseudo-residual in classification task?

14 XGBoost

14.1 Vanilla XGBoost

14.1.1 Definition

$$\widehat{y} = \sum_{t=1}^{T} f_t(x)$$

where $x \in \mathbb{R}^d$ represents the input feature vector, $y \in \mathbb{R}$ represents the real scalar prediction, T is the number of sequential additive functions (weak learners) and f_t corresponds to the independent weak learner (regression tree) of t-th additive round.

14.1.2 Learning

Objective. Consider the training process of t-th additive iteration (analyze the status before and after t-th training):

- Input:
  - dataset: $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{N}$ $(x^{(i)} \in \mathbb{R}^d, y^{(i)} \in \mathbb{R})$.
  - weak learners (e.g., CART) of first $t-1$ iterations: $\{f_k(\cdot)\}_{k=1}^{t-1}$.
  - strong learner after first $t-1$ iterations: $\widehat{y}_{t-1} = \sum_{k=1}^{t-1} f_k(x)$.
- Output:
  - t-th weak learner: $f_t(x)$
  - strong learner after first t iterations: $\widehat{y}_t = \sum_{k=1}^{t} f_k(x) = \widehat{y}_{t-1} + f_t(x)$.

Therefore we can formulate the optimization problem of the t-th iteration as follow:

$$\mathcal{J}^{(t)} = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}\left(\widehat{y}_t^{(i)}, y^{(i)}\right) + \Omega(\widehat{y}_t^{(i)})$$

Let's make a derivation of above objective following:

$$\begin{aligned}
&\min_{f_t \in \mathcal{F}} \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}\left(\widehat{y}_t^{(i)}, y^{(i)}\right) + \Omega(\widehat{y}_t) \\
\Rightarrow &\min_{f_t \in \mathcal{F}} \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}\left(\widehat{y}_{t-1}^{(i)} + f_t(x^{(i)}), y^{(i)}\right) + \Omega\left(\sum_{k=1}^{t} f_k\right) \quad (\because\ \widehat{y}_t = \sum_{k=1}^{t} f_k(x)) \\
\Rightarrow &\min_{f_t \in \mathcal{F}} \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}\left(\widehat{y}_{t-1}^{(i)} + f_t(x^{(i)}), y^{(i)}\right) + \Omega\left(f_t + \widehat{y}_{t-1}\right) \\
\Rightarrow &\min_{f_t \in \mathcal{F}} \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}\left(\widehat{y}_{t-1}^{(i)} + f_t(x^{(i)}), y^{(i)}\right) + \Omega\left(f_t\right) + \Omega\left(\widehat{y}_{t-1}\right) \quad (\because\ f_t \text{ and } \widehat{y}_{t-1} \text{ are independent with each other.}) \\
\Rightarrow &\min_{f_t \in \mathcal{F}} \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}\left(\widehat{y}_{t-1}^{(i)} + f_t(x^{(i)}), y^{(i)}\right) + \Omega\left(f_t\right) \quad (\because\ \Omega\left(\widehat{y}_{t-1}\right) \text{ is a constant.})
\end{aligned}$$

Theorem 14.1 (Taylor's Theorem)**.** *Taylor second-order expansion:*

$$f(x + \Delta x) \simeq f(x) + f'(x)\Delta x + \frac{1}{2} f''(x) \Delta x^2$$

Therefore perform Taylor second-order expansion for $\mathcal{L}$, we can get expanded second-order approximation loss function as follow:

$$\mathcal{L}\left(\widehat{y}_{t-1} + f_t(x), y\right) \simeq \mathcal{L}\left(\widehat{y}_{t-1}, y\right) + \frac{\partial \mathcal{L}\left(\widehat{y}_{t-1}, y\right)}{\partial f_t(x)} f_t(x) + \frac{1}{2} \frac{\partial \mathcal{L}\left(\widehat{y}_{t-1}, y\right)}{\partial^2 f_t(x)} f_t^2(x)$$

If it's hard to understand, consider the squared loss function:

$$\begin{aligned}\frac{\partial \mathcal{L}\left(\widehat{y}_{t-1}, y\right)}{\partial \widehat{y}_{t-1}} &= \frac{\partial}{\partial \widehat{y}_{t-1}} \frac{1}{2}\left(\widehat{y}_{t-1}-y\right)^2 \\ &= \widehat{y}_{t-1}-y \\ \frac{\partial \mathcal{L}\left(\widehat{y}_{t-1}, y\right)}{\partial^2 \widehat{y}_{t-1}} &= 1\end{aligned}$$

Bring second-order expansion in the above objective:

$$\min_{f_t \in \mathcal{F}} \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}\left(\widehat{y}_{t-1}^{(i)} + f_t(x^{(i)}), y^{(i)}\right) + \Omega\left(f_t\right)$$

$$\Rightarrow \min_{f_t \in \mathcal{F}} \frac{1}{N}\sum_{i=1}^{N} \left(\mathcal{L}\left(\widehat{y}_{t-1}^{(i)}, y^{(i)}\right) + \frac{\partial \mathcal{L}\left(\widehat{y}_{t-1}^{(i)}, y^{(i)}\right)}{\partial f_t(x^{(i)})} f_t(x^{(i)}) + \frac{1}{2}\frac{\partial \mathcal{L}\left(\widehat{y}_{t-1}^{(i)}, y^{(i)}\right)}{\partial^2 f_t(x^{(i)})} f_t^2(x^{(i)})\right) + \Omega\left(f_t\right)$$

$$\Rightarrow \min_{f_t \in \mathcal{F}} \frac{1}{N}\sum_{i=1}^{N} \left(\frac{\partial \mathcal{L}\left(\widehat{y}_{t-1}^{(i)}, y^{(i)}\right)}{\partial f_t(x^{(i)})} f_t(x^{(i)}) + \frac{1}{2}\frac{\partial \mathcal{L}\left(\widehat{y}_{t-1}^{(i)}, y^{(i)}\right)}{\partial^2 f_t(x^{(i)})} f_t^2(x^{(i)})\right) + \Omega\left(f_t\right) \quad (\because\ \mathcal{L}\left(\widehat{y}_{t-1}^{(i)}, y^{(i)}\right) \text{ is a constant.})$$

$$\Rightarrow \min_{f_t \in \mathcal{F}} \frac{1}{N}\sum_{i=1}^{N} \left(g_{t-1}^{(i)} f_t(x^{(i)}) + \frac{1}{2} h_{t-1}^{(i)} f_t^2(x^{(i)})\right) + \Omega\left(f_t\right) \quad (g_{t-1}^{(i)} \leftarrow \frac{\partial \mathcal{L}\left(\widehat{y}_{t-1}^{(i)}, y^{(i)}\right)}{\partial f_t(x^{(i)})}, h_{t-1}^{(i)} \leftarrow \frac{\partial \mathcal{L}\left(\widehat{y}_{t-1}^{(i)}, y^{(i)}\right)}{\partial^2 f_t(x^{(i)})})$$

Note that we use second-order expansion for loss function in XGBoost, but only use first-order expansion for loss function in GBM.

Problem: Why need to perform Taylor expansion for loss function $\mathcal{L}$?

- Theoretical Perspective:
  - know what we are learning, convergence.
- Engineering Perspective:
  - g_{t-1} and h_{t-1} comes from definition of specific loss function $\mathcal{L}$ (e.g., squared loss for regression, logit loss for classification).
  - The learning of function f_t only depends on the objective via g_{t-1} and h_{t-1}.
  - Think of how you can separate modules of your code when you are asked to implement boosted tree for both square loss and logistic loss.

Now we have two tricky problems to be solved immediately:

- How to formulate training loss further: $g_{t-1} f_t(x) + \frac{1}{2} h_{t-1} f_t^2(x)$?
- How to formulate regularization further: $\Omega(f_t)$?

Proposition 14.1 (Refine the Definition of Tree)**.** *We define a tree model f_t by a vector of scores in leaves, and a leaf index mapping function that maps an instance x to a leaf $q(x)$:*

$$f_t(x) = w_{q(x)},\ w \in \mathbb{R}^L,\ q : \mathbb{R}^d \to \{1, \ldots, L\}$$

Proposition 14.2 (Define Complexity of a Tree)**.** *Define complexity as (this is not the only possible definition):*

$$\Omega(f_t) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{L} w_j^2$$

where γT controls the number of leaves, $\frac{1}{2}\lambda \sum_{j=1}^{L} w_j^2$ controls weights of leaves.

Therefore, we can make futher derivation of learning objective:

$$\min_{f_t \in \mathcal{F}} \frac{1}{N}\sum_{i=1}^{N} \left(g_{t-1}^{(i)} f_t(x^{(i)}) + \frac{1}{2} h_{t-1}^{(i)} f_t^2(x^{(i)})\right) + \Omega\left(f_t\right)$$

$$\Rightarrow \min_{f_t \in \mathcal{F}} \frac{1}{N} \sum_{i=1}^{N} \left(g_{t-1}^{(i)} w_{q(x^{(i)})} + \frac{1}{2} h_{t-1}^{(i)} w_{q(x^{(i)})}^2 \right) + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^{L} w_j^2 \quad (\because \ f_t(x) = w_{q(x)}, \ \Omega(f_t) = \gamma T + \frac{1}{2} \lambda \sum_{j=1}^{L} w_j^2)$$

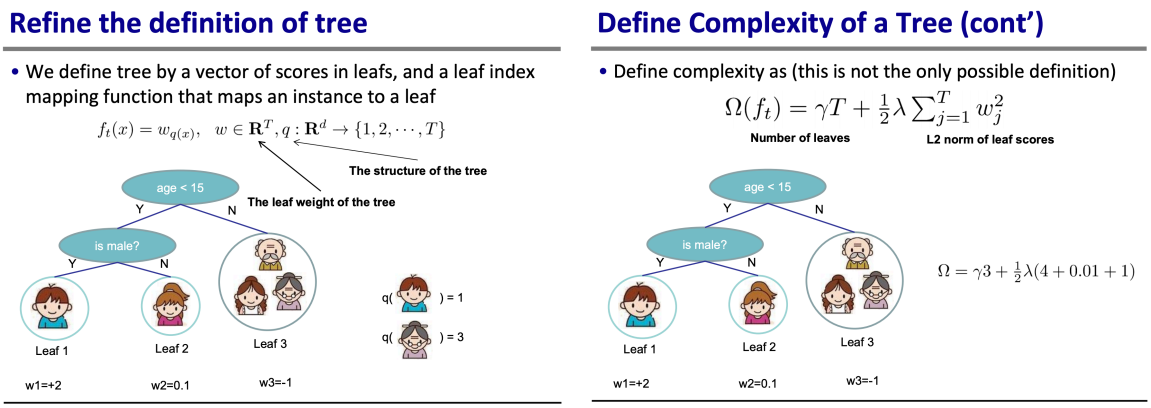


Figure 5: Refine the definition of tree and define complexity of a tree. Source: https://homes.cs.washington.edu/~tqchen/data/pdf/BoostedTree.pdf.

Because all samples are distributed on all leaves' nodes of the tree structure, therefore we can make following derivation:

$$\begin{aligned}
&\min_{f_t \in \mathcal{F}} \frac{1}{N} \sum_{i=1}^{N} \left(g_{t-1}^{(i)} w_{q(x^{(i)})} + \frac{1}{2} h_{t-1}^{(i)} w_{q(x^{(i)})}^2 \right) + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^{L} w_j^2 \\
\Rightarrow &\min_{f_t \in \mathcal{F}} \frac{1}{N} \sum_{j=1}^{L} \sum_{i \in \mathcal{I}_j} \left(g_{t-1}^{(i)} w_j + \frac{1}{2} h_{t-1}^{(i)} w_j^2 \right) + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^{L} w_j^2 \quad \text{(most important step!)} \\
\Rightarrow &\min_{f_t \in \mathcal{F}} \sum_{j=1}^{L} \sum_{i \in \mathcal{I}_j} \left(\frac{g_{t-1}^{(i)}}{N} w_j + \frac{1}{2} \frac{h_{t-1}^{(i)}}{N} w_j^2 \right) + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^{L} w_j^2 \\
\Rightarrow &\min_{f_t \in \mathcal{F}} \sum_{j=1}^{L} \left(\left(\sum_{i \in \mathcal{I}_j} \frac{g_{t-1}^{(i)}}{N} \right) w_j + \frac{1}{2} \left(\sum_{i \in \mathcal{I}_j} \frac{h_{t-1}^{(i)}}{N} \right) w_j^2 \right) + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^{L} w_j^2 \\
\Rightarrow &\min_{f_t \in \mathcal{F}} \sum_{j=1}^{L} \left(\left(\sum_{i \in \mathcal{I}_j} \frac{g_{t-1}^{(i)}}{N} \right) w_j + \frac{1}{2} \left(\sum_{i \in \mathcal{I}_j} \frac{h_{t-1}^{(i)}}{N} + \lambda \right) w_j^2 \right) + \gamma T \\
\Rightarrow &\min_{f_t \in \mathcal{F}} \sum_{j=1}^{L} \left(G_{t-1}^{(j)} w_j + \frac{1}{2} \left(H_{t-1}^{(j)} + \lambda \right) w_j^2 \right) + \gamma T \quad (G_{t-1}^{(j)} \leftarrow \sum_{i \in \mathcal{I}_j} \frac{g_{t-1}^{(i)}}{N}, \ H_{t-1}^{(j)} \leftarrow \sum_{i \in \mathcal{I}_j} \frac{h_{t-1}^{(i)}}{N})
\end{aligned}$$

Theorem 14.2 (Quadratic Convex Optimization)**.** *For the quadratic convex optimization problem:*

$$\min_x \ \mathcal{J}(x) = Gx + \frac{1}{2} H x^2$$

the optimal solution is:

$$x^* = -\frac{G}{H}$$

and corresponding optimal objective value is:

$$\mathcal{J}(x^*) = -\frac{1}{2} \frac{G^2}{H}$$

Because all leaves of a tree are independent of each other, therefore the above optimization problem can be considered as L independent optimization problems, and the optimal solution of j-th leaf is:

$$w_j^* = -\frac{G_{t-1}^{(j)}}{H_{t-1}^{(j)} + \lambda}$$

and the corresponding optimal value of j-th leaf is:

$$\mathcal{J}_j^{(t)*} = -\frac{1}{2}\frac{\left(G_{t-1}^{(j)}\right)^2}{H_{t-1}^{(j)} + \lambda} + \gamma T$$

Even though the optimal value of a single leaf node is known now, the optimal value of the original objective $\mathcal{J}^{(t)}$ is still unknown, because we are not able to determine the tree structure f_t, and the types of tree structure q are infinite. Therefore, we use a greedy strategy to find the best tree structure.

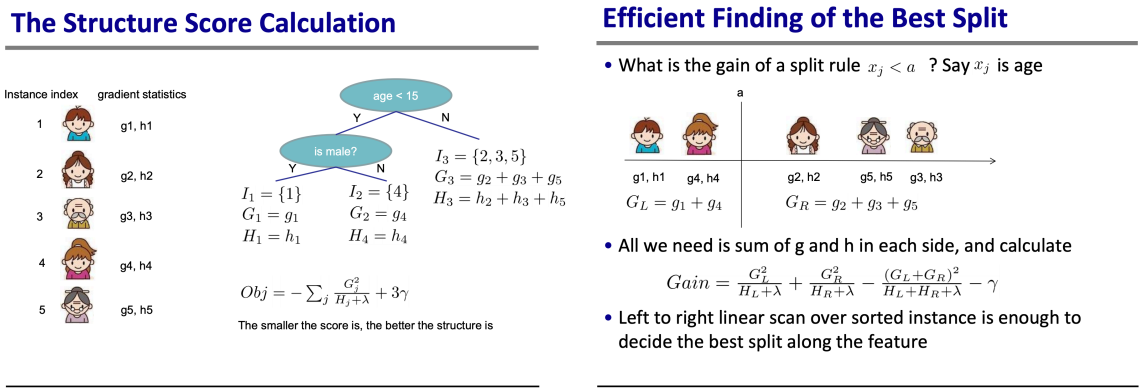


Figure 6: The structure score calculation and efficient finding of the best split in XGBoost. Source: https://homes.cs.washington.edu/~tqchen/data/pdf/BoostedTree.pdf.

Optimization. We build a greedy binary tree to optimize the model in each step.

Problem: How to find the best tree structure ? In practice, **grow the binary tree greedily**:

1. Start from binary tree with depth 0;
2. For each leaf node of the tree, try to add a split. The change of objective after adding the split is:

$$\begin{aligned} Gain &= \mathcal{J}_{\text{before-split}} - \mathcal{J}_{\text{after-split}} \\ &= \left(-\frac{1}{2}\frac{(G_\text{L}+G_\text{R})^2}{H_\text{L}+H_\text{R}+\lambda} + \gamma\right) - \left(-\frac{1}{2}\frac{G_\text{L}^2}{H_\text{L}+\lambda} + \gamma\right) - \left(-\frac{1}{2}\frac{G_\text{R}^2}{H_\text{R}+\lambda} + \gamma\right) \\ &= -\frac{1}{2}\frac{(G_\text{L}+G_\text{R})^2}{H_\text{L}+H_\text{R}+\lambda} + \gamma + \frac{1}{2}\frac{G_\text{L}^2}{H_\text{L}+\lambda} - \gamma + \frac{1}{2}\frac{G_\text{R}^2}{H_\text{R}+\lambda} - \gamma \\ &= \frac{1}{2}\left(\frac{G_\text{L}^2}{H_\text{L}+\lambda} + \frac{G_\text{R}^2}{H_\text{R}+\lambda} - \frac{(G_\text{L}+G_\text{R})^2}{H_\text{L}+H_\text{R}+\lambda}\right) - \gamma \end{aligned}$$

Problem: How to find the best split point for a leaf ? Enumerate over all features:

- For each feature, sort the instances by feature value. Note all features must be either numerical or categorical:
  - for numerical feature, sort by value directly;
  - for categorical feature, encode the categorical feature into numerical vector using one-hot encoding, each dimension of one-hot encoding vector means one-to-many pair.

- Use a linear scan to decide the best split along that feature
- Take the best split solution along all the features

Algorithm 15: Basic exact greedy algorithm for split finding[39].

Input: $\mathcal{I}^{M\times d}$: Instances set of current node (d is the depth of feature vector after one-hot enccoding).
Output: Split with max score.

$score \leftarrow -\inf$
$G \leftarrow \sum_{i\in\mathcal{I}} g^{(i)},\ H \leftarrow \sum_{i\in\mathcal{I}} h^{(i)}$
for $j = \{1, \ldots, d\}$ **do**
  $G_L \leftarrow 0,\ H_L \leftarrow 0$
  for *i in sorted($\mathcal{I}$, by x_j)* **do**
    $G_L \leftarrow G_L + g^{(i)},\ H_L \leftarrow H_L + h^{(i)}$
    $G_R \leftarrow G - G_L,\ H_R \leftarrow H - H_L$
    $score \leftarrow \max\left(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda}\right)$
  end
end
return Split with max score

Algorithm 16: Approximate algorithm for split finding[39].

for $k = \{1, \ldots d\}$ **do**
  Propose $\mathcal{S}_k = \{s_{k1}, s_{k2}, \ldots, s_{kl}\}$ by percentiles on feature k. ▷ d is the depth of raw feature vector (without one-hot encoding)
  Proposal can be done per tree (global), or per split (local).
end
for $k = \{1, \ldots d\}$ **do**
  $G_{kv} \leftarrow \sum_{i\in\{i|s_{k,v-1}<x_k^{(i)}\le s_{k,v}\}} g^{(i)}$ ▷ Summary first-order gradient inside a specific feature value bin
  $H_{kv} \leftarrow \sum_{i\in\{i|s_{k,v-1}<x_k^{(i)}\le s_{k,v}\}} h^{(i)}$ ▷ Summary second-order gradient inside a specific feature value bin
end
Follow same step as in previous section to find max score only among proposed splits.

Pruning and Regularization. Recall the gain when split a leaf node:

$$Gain = \frac{1}{2}\left(\frac{G_{\mathrm{L}}^2}{H_{\mathrm{L}}+\lambda} + \frac{G_{\mathrm{R}}^2}{H_{\mathrm{R}}+\lambda} - \frac{(G_{\mathrm{L}}+G_{\mathrm{R}})^2}{H_{\mathrm{L}}+H_{\mathrm{R}}+\lambda}\right) - \gamma$$

The higher the value of $Gain$, the better the split effect, but the value of $Gain$ would be negative. Therefore two pruning strategies (i.e., pre-stopping & post-prunning) both consider whether the split has negative gain.

- Pre-stopping: Stop split if the current $score$ (gain) on a specific feature j have negative gain, but maybe a split with negative $score$ now can benefit future splits.
- Post-Prunning: Grow a tree to maximum depth, recursively prune all the leaf splits with negative gain.

14.1.3 Cognition

Opinions. Some cognitions (future research works) on how to improve the model show following:

1. Replace uniform distribution of training dataset ($1/N$) by sample importance distribution ($\varphi(x, y; f_{t-1})$) (e.g., according to some kind of residual).
2. Using reinforcement learning methods for pruning.

15 LightGBM

16 CatBoost

Part IV
Deep Learning Algorithms

17 Feed-Forward Neural Networks (FFNNs)

18 Convolutional Neural Networks (CNNs)

19 Recurrent Neural Networks (RNNs)

20 Attention Mechanism

20.1 Vanilla Self-Attention

20.1.1 Motivation

Sequence encoding based on convolutional neural networks or recurrent neural networks both can be viewed as local coding methods, therefore only the local dependencies of the input information are modeled. Although a recurrent neural network (e.g., LSTM) can theoretically establish long-distance dependencies, due to the capacity of information delivery and the problem of vanishing gradient, in fact, only short-distance dependencies can be established. Therefore, we need a global encoding method for establishing long-distance dependencies.

There exists two main methods for establishing long-distance dependencies among input sequences:

- Increasing the number of layers in the network and use a deep network to obtain long-distance information interactions (e.g., multi-layer LSTMs)
- Using fully connected network.

A fully connected network is a very direct method for modeling long-distance dependencies of an input sequence. but it can't handle variable-length input sequences. Different input lengths have different connection weights $\boldsymbol{W}$. At this time, we can use the attention mechanism to "dynamically" generate different connection weights. This is the self-attention model.

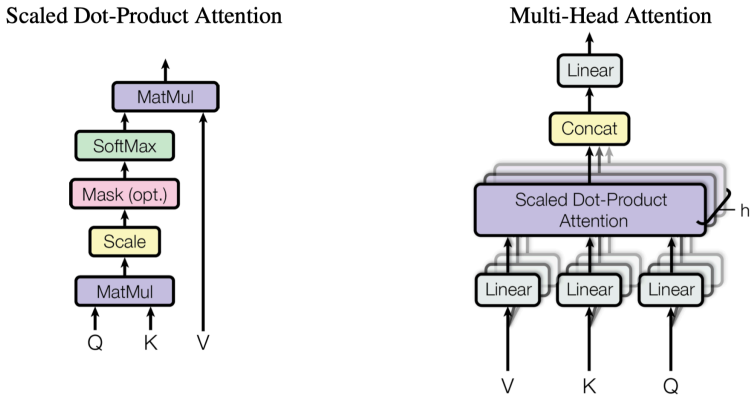


Figure 7: (left) Scaled dot-product attention. (right) Multi-head attention consists of several attention layers running in parallel. Source: (Vaswani et al., 2017)[11].

20.1.2 Definition

Methodology. To improve model capabilities, self-attention models often use a query-key-value mode, the calculation process is shown in Figure 9.

Assume we get the input sequence $\boldsymbol{X} = \left(\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \cdots, \boldsymbol{x}^{(N)}\right)^{\mathrm{T}} \in \mathbb{R}^{N \times d_{\mathcal{X}}}$, and we want to get the output sequence $\boldsymbol{Z} = \left(\boldsymbol{z}^{(1)}, \boldsymbol{z}^{(2)}, \cdots, \boldsymbol{z}^{(N)}\right)^{\mathrm{T}} \in \mathbb{R}^{N \times d_{\mathcal{V}}}$, the specific calculation process of the self-attention model is as follows:

1. For each input embedding $\boldsymbol{x}^{(i)} \in \mathbb{R}^{d_{\mathcal{X}}}$, We first map it linearly to three different spaces $\mathcal{Q}$, $\mathcal{K}$ and $\mathcal{V}$, and get corresponding query vector $\boldsymbol{q}^{(i)} \in \mathbb{R}^{d_{\mathcal{Q}}}$, key vector $\boldsymbol{k}^{(i)} \in \mathbb{R}^{d_{\mathcal{K}}}$ and value vector $\boldsymbol{v}^{(i)} \in \mathbb{R}^{d_{\mathcal{V}}}$. For the entire input sequence $\boldsymbol{X}$, the linear mapping process can be abbreviated as follows:

$$\boldsymbol{Q} = \left(\boldsymbol{q}^{(1)}, \boldsymbol{q}^{(2)}, \cdots, \boldsymbol{q}^{(N)}\right)^{\mathrm{T}} = \boldsymbol{X}\boldsymbol{W}_{\mathrm{Q}} \in \mathbb{R}^{N \times d_{\mathcal{Q}}}$$

$$\boldsymbol{K} = \left(\boldsymbol{k}^{(1)}, \boldsymbol{k}^{(2)}, \cdots, \boldsymbol{k}^{(N)}\right)^{\mathrm{T}} = \boldsymbol{X}\boldsymbol{W}_{\mathrm{K}} \in \mathbb{R}^{N \times d_{\mathcal{K}}}$$

$$\boldsymbol{V} = \left(\boldsymbol{v}^{(1)}, \boldsymbol{v}^{(2)}, \cdots, \boldsymbol{v}^{(N)}\right)^{\mathrm{T}} = \boldsymbol{X}\boldsymbol{W}_{\mathrm{V}} \in \mathbb{R}^{N \times d_{\mathcal{V}}}$$

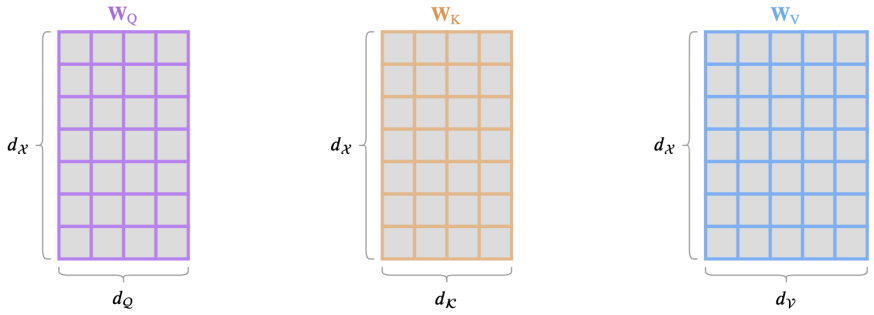


Figure 8: Parameter matrixs of linear mapping from $\mathcal{X}$ to $\mathcal{Q}$, $\mathcal{K}$ and $\mathcal{V}$ respectively. Note the distinction between dimension of original embedding space $d_{\mathcal{X}}$ and length of sequence N.

where shown in Figure 8:

- $\boldsymbol{W}_{\mathrm{Q}} \in \mathbb{R}^{d_{\mathcal{X}} \times d_{\mathcal{Q}}}$: Parameters matrix of linear mapping from original embedding space $\mathcal{X}$ to query space $\mathcal{Q}$.
- $\boldsymbol{W}_{\mathrm{K}} \in \mathbb{R}^{d_{\mathcal{X}} \times d_{\mathcal{K}}}$: Parameters matrix of linear mapping from original embedding space $\mathcal{X}$ to key space $\mathcal{K}$.
- $\boldsymbol{W}_{\mathrm{V}} \in \mathbb{R}^{d_{\mathcal{X}} \times d_{\mathcal{V}}}$: Parameters matrix of linear mapping from original embedding space $\mathcal{X}$ to value space $\mathcal{V}$.

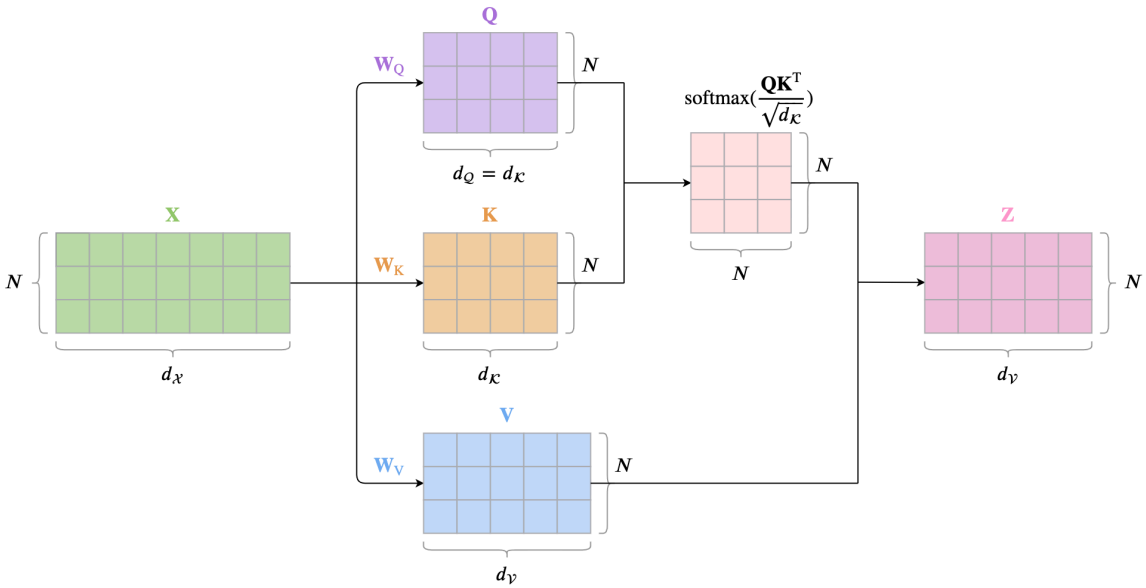


Figure 9: Calculation process of the self-attention. Note that $d_{\mathcal{Q}} = d_{\mathcal{K}}$.

2. For each query vector $\boldsymbol{q}^{(i)} \in \boldsymbol{Q}$, using key-value pair attention mechanism, get the output vector $\boldsymbol{z}^{(i)}$:

$$\begin{aligned}\boldsymbol{z}^{(i)} &= \mathrm{att}\Big((\boldsymbol{K}, \boldsymbol{V}), \boldsymbol{q}^{(i)}\Big) \\ &= \sum_{j=1}^{N} \alpha_{ij} \boldsymbol{v}^{(j)} \\ &= \sum_{j=1}^{N} \mathrm{softmax}\big(\mathrm{score}(\boldsymbol{k}^{(j)}, \boldsymbol{q}^{(i)})\big) \boldsymbol{v}^{(j)}\end{aligned}$$

where $j, i \in \{1, \cdots, N\}$ represent the position of input and output vector sequences respectively, α_{ij} indicates the weight that the i^{th} input focuses on the j^{th} input.

If we use a scaled dot product as an attention scoring function, the output vector sequence can be abbreviated as

$$\boldsymbol{Z} = \text{softmax}(\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}})\boldsymbol{V} \in \mathbb{R}^{N \times d_{\mathcal{V}}}$$

where $\text{softmax}(\cdot)$ is a function normalized by row here.

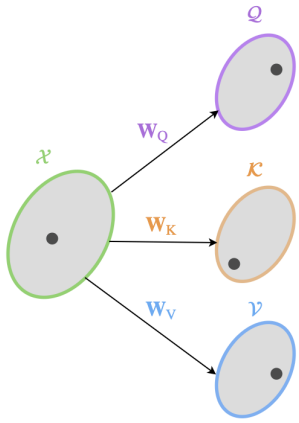


Figure 10: Mapping functions from original word embedding space $\mathcal{X}$ to query embedding space $\mathcal{Q}$, key embedding space $\mathcal{K}$ and value embedding space $\mathcal{V}$.

Let's make a expansion of above matrix operations for a more detailed explanation:

$$\begin{aligned}
\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}} &= \left(\boldsymbol{q}^{(1)}, \cdots, \boldsymbol{q}^{(i)}, \cdots, \boldsymbol{q}^{(N)}\right)^{\mathrm{T}} \left(\boldsymbol{k}^{(1)}, \cdots, \boldsymbol{k}^{(j)}, \cdots, \boldsymbol{k}^{(N)}\right) \\
&= \begin{pmatrix} (\boldsymbol{q}^{(1)})^{\mathrm{T}} \\ \vdots \\ (\boldsymbol{q}^{(i)})^{\mathrm{T}} \\ \vdots \\ (\boldsymbol{q}^{(N)})^{\mathrm{T}} \end{pmatrix} \left(\boldsymbol{k}^{(1)}, \cdots, \boldsymbol{k}^{(j)}, \cdots, \boldsymbol{k}^{(N)}\right) \\
&= \begin{pmatrix} \boldsymbol{q}^{(1)} \cdot \boldsymbol{k}^{(1)} & \cdots & \boldsymbol{q}^{(1)} \cdot \boldsymbol{k}^{(j)} & \cdots & \boldsymbol{q}^{(1)} \cdot \boldsymbol{k}^{(N)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \boldsymbol{q}^{(i)} \cdot \boldsymbol{k}^{(1)} & \cdots & \boldsymbol{q}^{(i)} \cdot \boldsymbol{k}^{(j)} & \cdots & \boldsymbol{q}^{(i)} \cdot \boldsymbol{k}^{(N)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \boldsymbol{q}^{(N)} \cdot \boldsymbol{k}^{(1)} & \cdots & \boldsymbol{q}^{(N)} \cdot \boldsymbol{k}^{(j)} & \cdots & \boldsymbol{q}^{(N)} \cdot \boldsymbol{k}^{(N)} \end{pmatrix} \in \mathbb{R}^{N \times N}
\end{aligned}$$

Note that $\boldsymbol{q}^{(i)} \cdot \boldsymbol{k}^{(j)} \neq \boldsymbol{q}^{(j)} \cdot \boldsymbol{k}^{(i)}$ and therefore $\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}$ is not a symmetric matrix. Where $\cdot$ operation means inner product between two vectors with same dimension:

$$\boldsymbol{q}^{(i)} \cdot \boldsymbol{k}^{(j)} = \sum_{\ell=1}^{d_{\mathcal{K}}} q_{\ell}^{(i)} k_{\ell}^{(j)}$$

and scale the inner product element-wise with $\sqrt{d_{\mathcal{K}}}$ we have

$$\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}} = \begin{pmatrix} \frac{\boldsymbol{q}^{(1)} \cdot \boldsymbol{k}^{(1)}}{\sqrt{d_{\mathcal{K}}}} & \cdots & \frac{\boldsymbol{q}^{(1)} \cdot \boldsymbol{k}^{(j)}}{\sqrt{d_{\mathcal{K}}}} & \cdots & \frac{\boldsymbol{q}^{(1)} \cdot \boldsymbol{k}^{(N)}}{\sqrt{d_{\mathcal{K}}}} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{\boldsymbol{q}^{(i)} \cdot \boldsymbol{k}^{(1)}}{\sqrt{d_{\mathcal{K}}}} & \cdots & \frac{\boldsymbol{q}^{(i)} \cdot \boldsymbol{k}^{(j)}}{\sqrt{d_{\mathcal{K}}}} & \cdots & \frac{\boldsymbol{q}^{(i)} \cdot \boldsymbol{k}^{(N)}}{\sqrt{d_{\mathcal{K}}}} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{\boldsymbol{q}^{(N)} \cdot \boldsymbol{k}^{(1)}}{\sqrt{d_{\mathcal{K}}}} & \cdots & \frac{\boldsymbol{q}^{(N)} \cdot \boldsymbol{k}^{(j)}}{\sqrt{d_{\mathcal{K}}}} & \cdots & \frac{\boldsymbol{q}^{(N)} \cdot \boldsymbol{k}^{(N)}}{\sqrt{d_{\mathcal{K}}}} \end{pmatrix} \in \mathbb{R}^{N \times N}$$

therefore we can obtain the result matrix after softmax operation by row

$$\text{softmax}(\frac{\boldsymbol{QK}^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}}) = \begin{pmatrix} \frac{\exp(\frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(1)}}{\sqrt{d_{\mathcal{K}}}})}{\sum_{\ell=1}^{N}\exp(\frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_{\mathcal{K}}}})} & \cdots & \frac{\exp(\frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(j)}}{\sqrt{d_{\mathcal{K}}}})}{\sum_{\ell=1}^{N}\exp(\frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_{\mathcal{K}}}})} & \cdots & \frac{\exp(\frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(N)}}{\sqrt{d_{\mathcal{K}}}})}{\sum_{\ell=1}^{N}\exp(\frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_{\mathcal{K}}}})} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{\exp(\frac{\boldsymbol{q}^{(i)}\cdot\boldsymbol{k}^{(1)}}{\sqrt{d_{\mathcal{K}}}})}{\sum_{\ell=1}^{N}\exp(\frac{\boldsymbol{q}^{(i)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_{\mathcal{K}}}})} & \cdots & \frac{\exp(\frac{\boldsymbol{q}^{(i)}\cdot\boldsymbol{k}^{(j)}}{\sqrt{d_{\mathcal{K}}}})}{\sum_{\ell=1}^{N}\exp(\frac{\boldsymbol{q}^{(i)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_{\mathcal{K}}}})} & \cdots & \frac{\exp(\frac{\boldsymbol{q}^{(i)}\cdot\boldsymbol{k}^{(N)}}{\sqrt{d_{\mathcal{K}}}})}{\sum_{\ell=1}^{N}\exp(\frac{\boldsymbol{q}^{(i)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_{\mathcal{K}}}})} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{\exp(\frac{\boldsymbol{q}^{(N)}\cdot\boldsymbol{k}^{(1)}}{\sqrt{d_{\mathcal{K}}}})}{\sum_{\ell=1}^{N}\exp(\frac{\boldsymbol{q}^{(N)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_{\mathcal{K}}}})} & \cdots & \frac{\exp(\frac{\boldsymbol{q}^{(N)}\cdot\boldsymbol{k}^{(j)}}{\sqrt{d_{\mathcal{K}}}})}{\sum_{\ell=1}^{N}\exp(\frac{\boldsymbol{q}^{(N)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_{\mathcal{K}}}})} & \cdots & \frac{\exp(\frac{\boldsymbol{q}^{(N)}\cdot\boldsymbol{k}^{(N)}}{\sqrt{d_{\mathcal{K}}}})}{\sum_{\ell=1}^{N}\exp(\frac{\boldsymbol{q}^{(N)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_{\mathcal{K}}}})} \end{pmatrix} \in \mathbb{R}^{N\times N}$$

where $\frac{\exp(\frac{\boldsymbol{q}^{(i)}\cdot\boldsymbol{k}^{(j)}}{\sqrt{d_{\mathcal{K}}}})}{\sum_{\ell=1}^{N}\exp(\frac{\boldsymbol{q}^{(i)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_{\mathcal{K}}}})} \in (0,1)$ **means the relative importance (effect) of j-th token on i-th token, compared among all tokens when encoding i-th token.** . Note that

$$\because \boldsymbol{q}^{(i)} \neq \boldsymbol{q}^{(j)}$$
$$\because \boldsymbol{k}^{(j)} \neq \boldsymbol{k}^{(i)}$$
$$\therefore \boldsymbol{q}^{(i)}\cdot\boldsymbol{k}^{(j)} \neq \boldsymbol{q}^{(j)}\cdot\boldsymbol{k}^{(i)}$$

Therefore $\boldsymbol{QK}^{\mathrm{T}}$, $\frac{\boldsymbol{QK}^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}}$ and $\text{softmax}(\frac{\boldsymbol{QK}^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}})$ are all asymmetric matrixs. Finally we can get the output sequence as follows:

$$\begin{aligned}\boldsymbol{Z} &= \text{softmax}(\frac{\boldsymbol{QK}^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}})\boldsymbol{V} \\ &= \begin{pmatrix} \frac{\exp(\frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(1)}}{\sqrt{d_{\mathcal{K}}}})}{\sum_{\ell=1}^{N}\exp(\frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_{\mathcal{K}}}})} & \cdots & \frac{\exp(\frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(j)}}{\sqrt{d_{\mathcal{K}}}})}{\sum_{\ell=1}^{N}\exp(\frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_{\mathcal{K}}}})} & \cdots & \frac{\exp(\frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(N)}}{\sqrt{d_{\mathcal{K}}}})}{\sum_{\ell=1}^{N}\exp(\frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_{\mathcal{K}}}})} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{\exp(\frac{\boldsymbol{q}^{(i)}\cdot\boldsymbol{k}^{(1)}}{\sqrt{d_{\mathcal{K}}}})}{\sum_{\ell=1}^{N}\exp(\frac{\boldsymbol{q}^{(i)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_{\mathcal{K}}}})} & \cdots & \frac{\exp(\frac{\boldsymbol{q}^{(i)}\cdot\boldsymbol{k}^{(j)}}{\sqrt{d_{\mathcal{K}}}})}{\sum_{\ell=1}^{N}\exp(\frac{\boldsymbol{q}^{(i)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_{\mathcal{K}}}})} & \cdots & \frac{\exp(\frac{\boldsymbol{q}^{(i)}\cdot\boldsymbol{k}^{(N)}}{\sqrt{d_{\mathcal{K}}}})}{\sum_{\ell=1}^{N}\exp(\frac{\boldsymbol{q}^{(i)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_{\mathcal{K}}}})} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{\exp(\frac{\boldsymbol{q}^{(N)}\cdot\boldsymbol{k}^{(1)}}{\sqrt{d_{\mathcal{K}}}})}{\sum_{\ell=1}^{N}\exp(\frac{\boldsymbol{q}^{(N)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_{\mathcal{K}}}})} & \cdots & \frac{\exp(\frac{\boldsymbol{q}^{(N)}\cdot\boldsymbol{k}^{(j)}}{\sqrt{d_{\mathcal{K}}}})}{\sum_{\ell=1}^{N}\exp(\frac{\boldsymbol{q}^{(N)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_{\mathcal{K}}}})} & \cdots & \frac{\exp(\frac{\boldsymbol{q}^{(N)}\cdot\boldsymbol{k}^{(N)}}{\sqrt{d_{\mathcal{K}}}})}{\sum_{\ell=1}^{N}\exp(\frac{\boldsymbol{q}^{(N)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_{\mathcal{K}}}})} \end{pmatrix} \begin{pmatrix} (\boldsymbol{v}^{(1)})^{\mathrm{T}} \\ \vdots \\ (\boldsymbol{v}^{(j)})^{\mathrm{T}} \\ \vdots \\ (\boldsymbol{v}^{(N)})^{\mathrm{T}} \end{pmatrix} \\ &= \begin{pmatrix} \sum_{j=1}^{N}\frac{\exp(\frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(j)}}{\sqrt{d_{\mathcal{K}}}})}{\sum_{\ell=1}^{N}\exp(\frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_{\mathcal{K}}}})}(\boldsymbol{v}^{(j)})^{\mathrm{T}} \\ \vdots \\ \sum_{j=1}^{N}\frac{\exp(\frac{\boldsymbol{q}^{(i)}\cdot\boldsymbol{k}^{(j)}}{\sqrt{d_{\mathcal{K}}}})}{\sum_{\ell=1}^{N}\exp(\frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_{\mathcal{K}}}})}(\boldsymbol{v}^{(j)})^{\mathrm{T}} \\ \vdots \\ \sum_{j=1}^{N}\frac{\exp(\frac{\boldsymbol{q}^{(N)}\cdot\boldsymbol{k}^{(j)}}{\sqrt{d_{\mathcal{K}}}})}{\sum_{\ell=1}^{N}\exp(\frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_{\mathcal{K}}}})}(\boldsymbol{v}^{(j)})^{\mathrm{T}} \end{pmatrix} = \begin{pmatrix} \sum_{j=1}^{N}\text{softmax}\big(\text{score}(\boldsymbol{k}^{(j)},\boldsymbol{q}^{(1)})\big)(\boldsymbol{v}^{(j)})^{\mathrm{T}} \\ \vdots \\ \sum_{j=1}^{N}\text{softmax}\big(\text{score}(\boldsymbol{k}^{(j)},\boldsymbol{q}^{(i)})\big)(\boldsymbol{v}^{(j)})^{\mathrm{T}} \\ \vdots \\ \sum_{j=1}^{N}\text{softmax}\big(\text{score}(\boldsymbol{k}^{(j)},\boldsymbol{q}^{(N)})\big)(\boldsymbol{v}^{(j)})^{\mathrm{T}} \end{pmatrix} \in \mathbb{R}^{N\times d_{\mathcal{V}}}\end{aligned}$$

where $\text{softmax}\big(\text{score}(\boldsymbol{k}^{(j)},\boldsymbol{q}^{(i)})\big) \in \mathbb{R}$ and $\boldsymbol{v}^{(j)} \in \mathbb{R}^{d_{\mathcal{V}}}$. For each vector in $\boldsymbol{Z}$, $\sum_{j=1}^{N}\frac{\exp(\frac{\boldsymbol{q}^{(i)}\cdot\boldsymbol{k}^{(j)}}{\sqrt{d_{\mathcal{K}}}})}{\sum_{l=1}^{N}\exp(\frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_{\mathcal{K}}}})}\boldsymbol{v}^{(j)} \in \mathbb{R}^{d_{\mathcal{V}}}$ integrates (i.e., weighted average sum) information from all terms into the i^{th} (query position i, not key position) term, but does not include position information of the sequence.

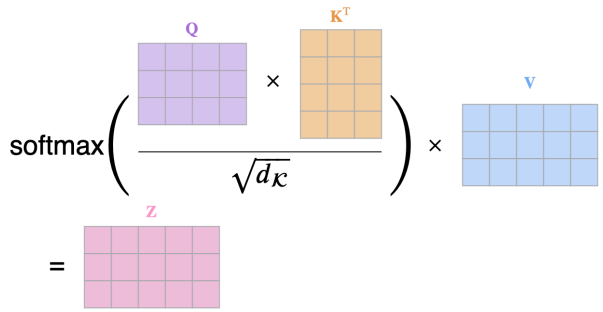


Figure 11: Self-Attention matrix operation interpretation. $\boldsymbol{Z} = \text{softmax}(\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}})\boldsymbol{V} \in \mathbb{R}^{N \times d_{\mathcal{V}}}$.

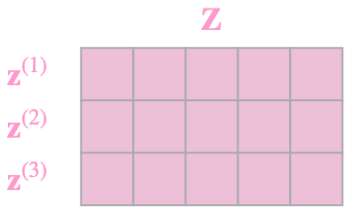


Figure 12: Output sequence $\boldsymbol{Z}$ stacks N output vectors $\boldsymbol{z}^{(i)} \in \mathbb{R}^{d_{\mathcal{V}}}$ row by row.

Each output vector $\boldsymbol{z}^{(i)}$ means the information encoding corresponding the i^{th} term (e.g., word) in a sequence, considering long-distance dependencies among the whole input sequence.

More detail of the operation process, for example, an input sequence "act gets results", all operations need to be done show as follows:

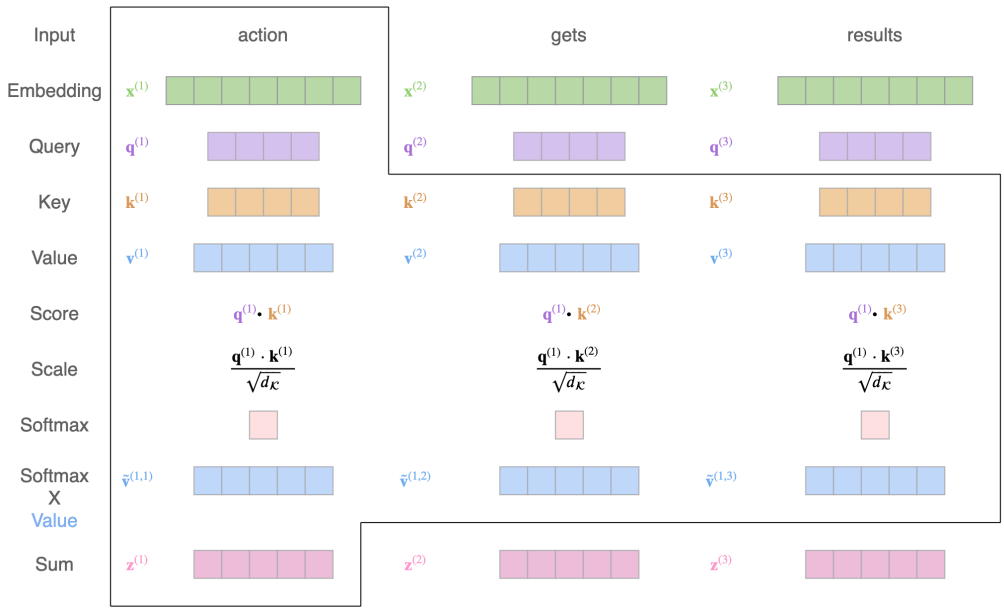


Figure 13: Self-attention operation of sequence "act gets results" with query word "action", which corresponds to $\left(\text{softmax}(\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}})\right)^{(1)}\boldsymbol{V} \in \mathbb{R}^{1 \times d_{\mathcal{V}}}$.

It corresponds to the computation process of 1-th row of $\text{softmax}(\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}})$ and $\boldsymbol{V}$:

$$\left(\text{softmax}(\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}})\right)^{(1)}\boldsymbol{V} = \begin{pmatrix} \frac{\exp(\frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(1)}}{\sqrt{d_{\mathcal{K}}}})}{\sum_{l=1}^{3}\exp(\frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_{\mathcal{K}}}})} & \cdots & \frac{\exp(\frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(j)}}{\sqrt{d_{\mathcal{K}}}})}{\sum_{l=1}^{3}\exp(\frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_{\mathcal{K}}}})} & \cdots & \frac{\exp(\frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(N)}}{\sqrt{d_{\mathcal{K}}}})}{\sum_{l=1}^{3}\exp(\frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_{\mathcal{K}}}})} \end{pmatrix} \begin{pmatrix} (\boldsymbol{v}^{(1)})^{\mathrm{T}} \\ \vdots \\ (\boldsymbol{v}^{(j)})^{\mathrm{T}} \\ \vdots \\ (\boldsymbol{v}^{(N)})^{\mathrm{T}} \end{pmatrix}$$

$$= \left(\sum_{j=1}^{3}\frac{\exp(\frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(j)}}{\sqrt{d_{\mathcal{K}}}})}{\sum_{l=1}^{3}\exp(\frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_{\mathcal{K}}}})}(\boldsymbol{v}^{(j)})^{\mathrm{T}}\right) = \left(\sum_{j=1}^{3}\text{softmax}\big(\text{score}(\boldsymbol{k}^{(j)}, \boldsymbol{q}^{(1)})\big)(\boldsymbol{v}^{(j)})^{\mathrm{T}}\right) \in \mathbb{R}^{1 \times d_{\mathcal{V}}}$$

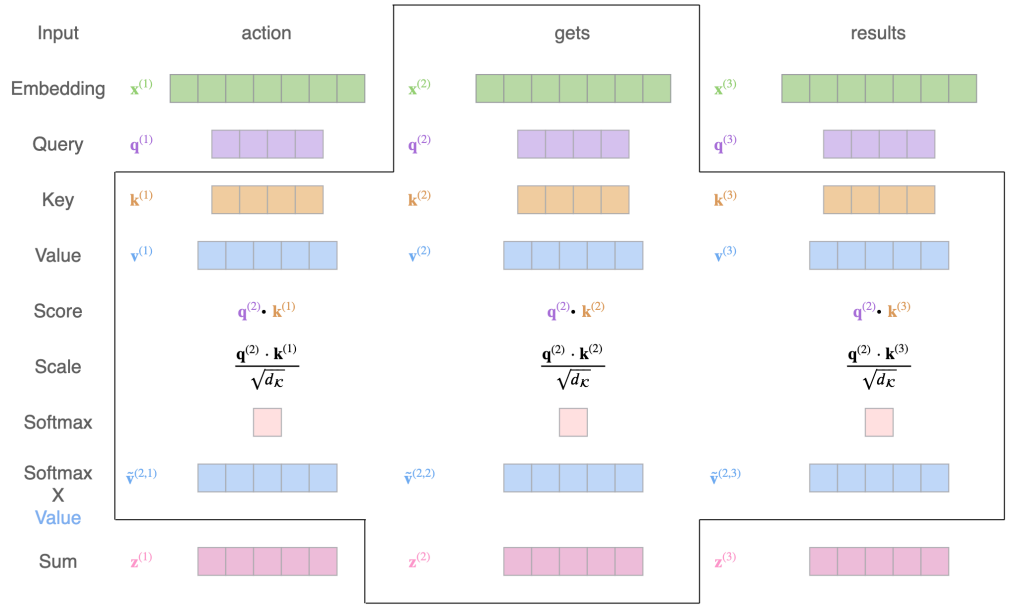


Figure 14: Self-attention operation of sequence "act gets results" with query word "gets", which corresponds to $\left(\text{softmax}(\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}})\right)^{(2)}\boldsymbol{V} \in \mathbb{R}^{1\times d_{\mathcal{V}}}$.

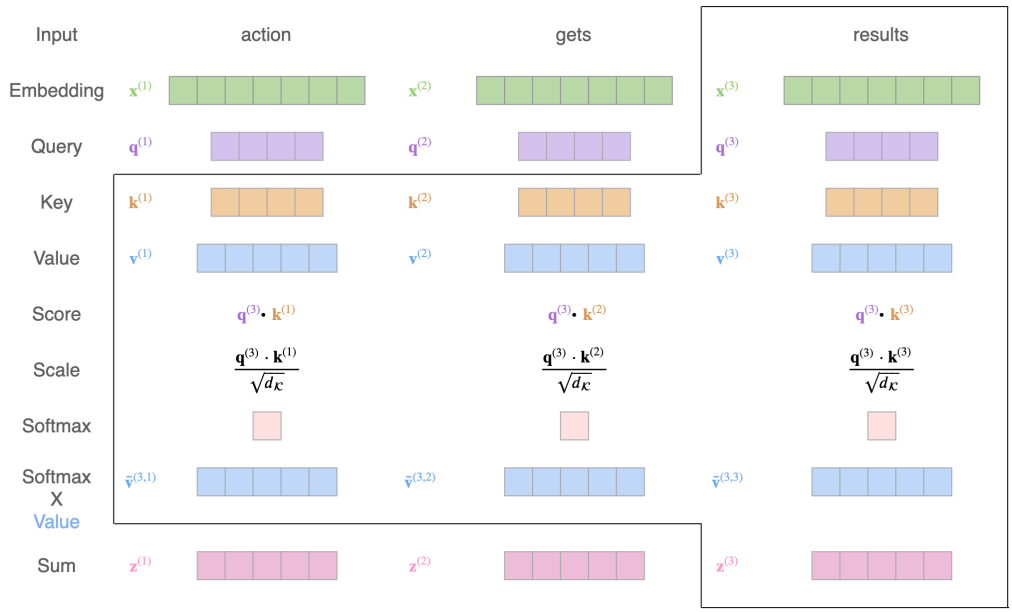


Figure 15: Self-attention operation of sequence "act gets results" with query word "results", which corresponds to $\left(\text{softmax}(\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}})\right)^{(3)}\boldsymbol{V} \in \mathbb{R}^{1\times d_{\mathcal{V}}}$.

Masked Self-Attention. Mask operation is an optional operation in self-attention computation process (See Figure 7). The computation process of masked self-attention is as follows (for example, with a length of sequence $N = 4$):

Figure 16: An example for interpreting masked self-attention mechanism. Source: https://jalammar.github.io/illustrated-gpt2/.

1. compute unnormalized attention matrix

$$\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}} = \begin{pmatrix} \frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(1)}}{\sqrt{d_{\mathcal{K}}}} & \frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(2)}}{\sqrt{d_{\mathcal{K}}}} & \frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(3)}}{\sqrt{d_{\mathcal{K}}}} & \frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(4)}}{\sqrt{d_{\mathcal{K}}}} \\ \frac{\boldsymbol{q}^{(2)}\cdot\boldsymbol{k}^{(1)}}{\sqrt{d_{\mathcal{K}}}} & \frac{\boldsymbol{q}^{(2)}\cdot\boldsymbol{k}^{(2)}}{\sqrt{d_{\mathcal{K}}}} & \frac{\boldsymbol{q}^{(2)}\cdot\boldsymbol{k}^{(3)}}{\sqrt{d_{\mathcal{K}}}} & \frac{\boldsymbol{q}^{(2)}\cdot\boldsymbol{k}^{(4)}}{\sqrt{d_{\mathcal{K}}}} \\ \frac{\boldsymbol{q}^{(3)}\cdot\boldsymbol{k}^{(1)}}{\sqrt{d_{\mathcal{K}}}} & \frac{\boldsymbol{q}^{(3)}\cdot\boldsymbol{k}^{(2)}}{\sqrt{d_{\mathcal{K}}}} & \frac{\boldsymbol{q}^{(3)}\cdot\boldsymbol{k}^{(3)}}{\sqrt{d_{\mathcal{K}}}} & \frac{\boldsymbol{q}^{(3)}\cdot\boldsymbol{k}^{(4)}}{\sqrt{d_{\mathcal{K}}}} \\ \frac{\boldsymbol{q}^{(4)}\cdot\boldsymbol{k}^{(1)}}{\sqrt{d_{\mathcal{K}}}} & \frac{\boldsymbol{q}^{(4)}\cdot\boldsymbol{k}^{(2)}}{\sqrt{d_{\mathcal{K}}}} & \frac{\boldsymbol{q}^{(4)}\cdot\boldsymbol{k}^{(3)}}{\sqrt{d_{\mathcal{K}}}} & \frac{\boldsymbol{q}^{(4)}\cdot\boldsymbol{k}^{(4)}}{\sqrt{d_{\mathcal{K}}}} \end{pmatrix} \in \mathbb{R}^{N\times N}$$

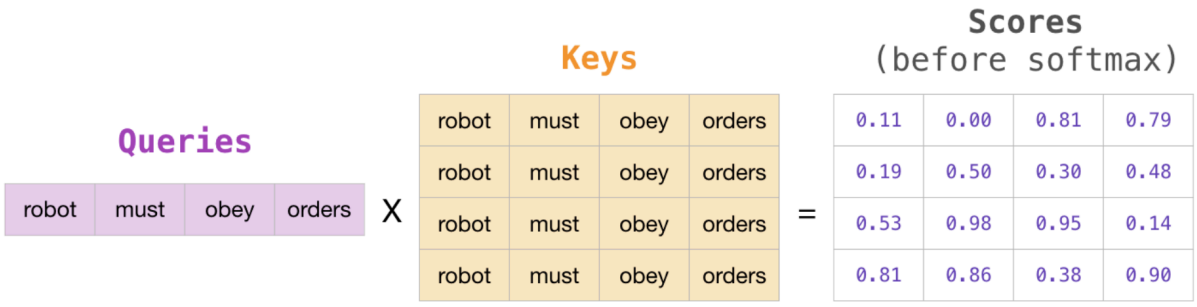


Figure 17: An example for interpreting masked self-attention mechanism, computing $\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}}$ (before softmax operation). Source: https://jalammar.github.io/illustrated-gpt2/.

2. get the masked attention score matrix $\mathrm{mask}(\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}}; \boldsymbol{M}) = \frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}} + \boldsymbol{M} \in \mathbb{R}^{N\times N}$ based on the prior knowledge, for example:

Figure 18: An example for interpreting masked self-attention mechanism, get the masked socres matrix $\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}} + \boldsymbol{M} \in \mathbb{R}^{N\times N}$ based on the piror knowledge. Source: https://jalammar.github.io/illustrated-gpt2/.

$$\boldsymbol{M} = \begin{pmatrix} 0 & 0 & -\inf & -\inf \\ -\inf & -\inf & 0 & -\inf \\ -\inf & -\inf & 0 & 0 \\ 0 & 0 & -\inf & -\inf \end{pmatrix} \in \mathbb{R}^{N \times N}$$

$$\text{mask}(\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}}; \boldsymbol{M}) = \frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}} + \boldsymbol{M}$$

$$= \begin{pmatrix} \frac{\boldsymbol{q}^{(1)} \cdot \boldsymbol{k}^{(1)}}{\sqrt{d_{\mathcal{K}}}} & \frac{\boldsymbol{q}^{(1)} \cdot \boldsymbol{k}^{(2)}}{\sqrt{d_{\mathcal{K}}}} & -\inf & -\inf \\ -\inf & -\inf & \frac{\boldsymbol{q}^{(2)} \cdot \boldsymbol{k}^{(3)}}{\sqrt{d_{\mathcal{K}}}} & -\inf \\ -\inf & -\inf & \frac{\boldsymbol{q}^{(3)} \cdot \boldsymbol{k}^{(3)}}{\sqrt{d_{\mathcal{K}}}} & \frac{\boldsymbol{q}^{(3)} \cdot \boldsymbol{k}^{(4)}}{\sqrt{d_{\mathcal{K}}}} \\ \frac{\boldsymbol{q}^{(4)} \cdot \boldsymbol{k}^{(1)}}{\sqrt{d_{\mathcal{K}}}} & \frac{\boldsymbol{q}^{(4)} \cdot \boldsymbol{k}^{(2)}}{\sqrt{d_{\mathcal{K}}}} & -\inf & -\inf \end{pmatrix} \in \mathbb{R}^{N \times N}$$

where $\mathrm{M}_{ij} \in \{0, -\inf\}$ means whether i-th token need to prevent the influence of the j-th token on it, if $\mathrm{M}_{ij} = -\inf$, prevent (i.e., mask out)! Else, allow j-th token's attendance on i-th token.

the reason why we use a very negative value as the value to replace the masked attention value is that, after softmax operation later, it can become a very tiny fraction (e.g., $-10^9 \rightarrow 0.00012$), namely being hardly relevant.

3. get the normalized (by row) masked attention scores matrix

$$\text{softmax}(\text{mask}(\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}}; \boldsymbol{M})) = \text{softmax}(\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}} + \boldsymbol{M}) \in \mathbb{R}^{N \times N}$$

where $\text{softmax}(\cdot)$ is an operation along row. What above scores table means is the following:

Masked Scores (before softmax)

0.11	-inf	-inf	-inf
0.19	0.50	-inf	-inf
0.53	0.98	0.95	-inf
0.81	0.86	0.38	0.90

Softmax (along rows) →

Scores

1	0	0	0
0.48	0.52	0	0
0.31	0.35	0.34	0
0.25	0.26	0.23	0.26

Figure 19: An example for interpreting the masked self-attention mechanism, get the masked socres matrix $\text{softmax}(\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}} + \boldsymbol{M}) \in \mathbb{R}^{N \times N}$ based on the piror knowledge. Source: https://jalammar.github.io/illustrated-gpt2/.

- When the model processes the first example in the dataset (row #1), which contains only one word ("robot"), 100% of its attention will be on that word.
- When the model processes the second example in the dataset (row #2), which contains the words ("robot must"), when it processes the word "must", 48% of its attention will be on "robot", and 52% of its attention will be on "must".
- And so on.

Implementation. A PyTorch implementation of self-attention (i.e., scaled dot product attention) as follows:

```
# Copyright (C) 2020 Tong Jia. All rights reserved.
import math
import torch
import torch.nn.functional as F


def subsequent_mask(size):
    """Mask out subsequent positions.

    Parameters
```

```
    ----------
    :param size: (int) -- size of matrix (e.g. d_k).
    :return: (Tensor) -- bool lower triangular tensor with shape (1, size, size).

    For example:
    subsequent_mask(size=7):
    tensor([[[ True, False, False, False, False, False, False],
             [ True,  True, False, False, False, False, False],
             [ True,  True,  True, False, False, False, False],
             [ True,  True,  True,  True, False, False, False],
             [ True,  True,  True,  True,  True, False, False],
             [ True,  True,  True,  True,  True,  True, False],
             [ True,  True,  True,  True,  True,  True,  True]]])
    """
    attn_shape = (1, size, size)
    subsequent_mask = np.triu(np.ones(attn_shape), k=1).astype('uint8')
    return torch.from_numpy(subsequent_mask) == 0


def attention(query, key, value, mask=None, dropout=None):
    """Compute 'Scaled Dot Product Attention'.

    Parameters
    ----------
    :param query: (Tensor) -- (BatchSize, N, d_k)
    :param key: (Tensor) -- (BatchSize, N, d_k)
    :param value: (Tensor) -- (BatchSize, N, d_v)
    :param mask: (Tensor) -- (BatchSize, N, N)
    :param dropout: (torch.Module)

    Returns
    ----------
    :return z_self_attn: (Tensor) -- (BatchSize, N, d_v)
    :return p_attn: (Tensor) -- (BatchSize, N, N)
    """
    d_k = query.size(-1)
    scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d_k)  # (BatchSize, N, N)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)
    p_attn = F.softmax(scores, dim = -1)  # (BatchSize, N, N)
    if dropout is not None:
        p_attn = dropout(p_attn)
    z_self_attn = torch.matmul(p_attn, value)  # (BatchSize, N, d_v)
    return z_self_attn, p_attn
```

20.1.3 Cognition

Opinions.

- **(∗ Open to question)**Because embedding of a sequence $\boldsymbol{X} \in \mathbb{R}^{N \times d_{\mathcal{X}}}$ need to be learned, also linear mapping matrix $\boldsymbol{W}_{\mathrm{Q}} \in \mathbb{R}^{d_{\mathcal{X}} \times d_{\mathcal{Q}}}$ from embedding space to query space, $\boldsymbol{W}_{\mathrm{K}} \in \mathbb{R}^{d_{\mathcal{X}} \times d_{\mathcal{K}}}$ from embedding space to key space and $\boldsymbol{W}_{\mathrm{V}} \in \mathbb{R}^{d_{\mathcal{X}} \times d_{\mathcal{V}}}$ from embedding space to value space need to be learned, to both protect the integrity of the information in the original embedded space $\mathcal{X}$, and keep pure directionality of the linear mapping, we suggest to:
  - normalize the parameter matrix $\boldsymbol{W}_{\mathrm{Q}}$ along each row.
  - normalize the parameter matrix $\boldsymbol{W}_{\mathrm{K}}$ along each row.
  - normalize the parameter matrix $\boldsymbol{W}_{\mathrm{V}}$ along each row.
- If using masked self-attention, the sum of each row is reduced after masking out, should we allocate all masked-column gap values before and after mask operation to the columns will not be masked proportionally, in order to keep sum of a row $(\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}})^{(i)}$ constantly both before and after masking columns $\{j \,|\, j \geq i\}$ out? (e.g., $12 + 135 + 1752 + 3851 \neq 12 + -10^9 + -10^9 + -10^9$ when encode 2-th (index starting from 1) token in encoder and decoder.)

Reading Materials.

- Jay Alammar: The Illustrated Transformer.
- How do Transformers Work in NLP? A Guide to the Latest State-of-the-Art Models.

20.2 Bi-Directional Block Self-Attention

20.2.1 Motivation

20.2.2 Definition

20.2.3 Cognition

20.3 Multi-Head Attention

20.3.1 Motivation

20.3.2 Definition

20.3.3 Cognition

21 Encoder-Decoder Networks

21.1 Transformer

21.1.1 Motivation

- Inherently sequential nature of recurrent models (e.g., RNNs, LSTMs, GRUs) precludes parallelization within training process[11].
- Attention mechanisms has still been used in conjunction with a recurrent network, even if it allowing modeling of dependencies without regard to their distance in the input or output sequences[11].

21.1.2 Contribution

- Presenting the Transformer model, the rst sequence transduction model based entirely on attention, replacing the recurrent layers most commonly used in encoder-decoder architectures with multi-head self-attention.
- Transformer can be trained signicantly faster than architectures based on recurrent or convolutional layers.

21.1.3 Definition

A High-Level Look. The Transformer follows this overall architecture using stacked self-attention and point-wise, fully connected layers for both the encoder and decoder, shown in the left and right halves of Figure 20, respectively.

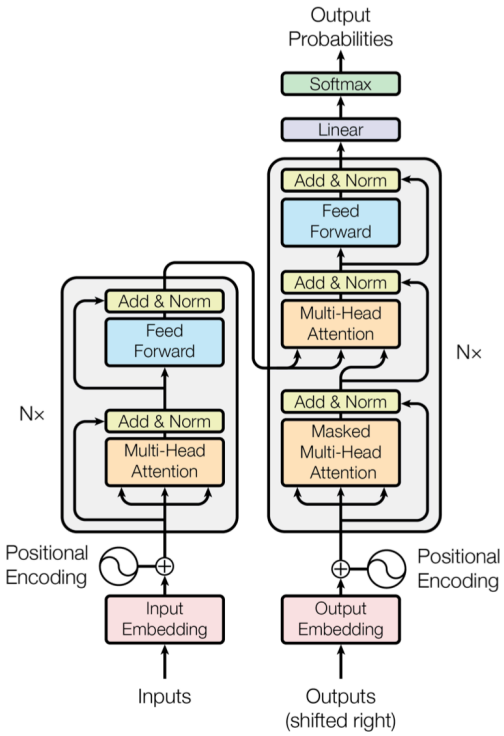


Figure 20: The Transformer model architecture. Source: (Vaswani et al., 2017)[11].

According to the different modules of the model, we will do the following introduction:

- self-attention
- multi-head attention
- residual connection & layer normalization
- feed forward neural network
- encoder

- masked multi-head attention
- encoder-decoder attention
- decoder
- positional encoding (Not Accomplished)

Self-Attention. See Subsection 20.1 and Figure 9 for details, a single head self-attention can be formulated as follows:

$$
\begin{aligned}
&\boldsymbol{Q} = \boldsymbol{X}\boldsymbol{W}_{\mathrm{Q}} \in \mathbb{R}^{N \times d_{\mathcal{Q}}} \\
&\boldsymbol{K} = \boldsymbol{X}\boldsymbol{W}_{\mathrm{K}} \in \mathbb{R}^{N \times d_{\mathcal{K}}} \quad (d_{\mathcal{Q}} = d_{\mathcal{K}}) \\
&\boldsymbol{V} = \boldsymbol{X}\boldsymbol{W}_{\mathrm{V}} \in \mathbb{R}^{N \times d_{\mathcal{V}}} \\
&\boldsymbol{Z} = \mathrm{softmax}\Big(\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}}\Big)\boldsymbol{V} \in \mathbb{R}^{N \times d_{\mathcal{V}}} \quad (\mathrm{softmax}(\cdot) \text{ is a function normalized by row here})
\end{aligned}
$$

where $\boldsymbol{X} \in \mathbb{R}^{N \times d_{\mathcal{X}}}$ is per-sample input sequence, and trainable parameters as follows:

- $\boldsymbol{W}_{\mathrm{Q}} \in \mathbb{R}^{d_{\mathcal{X}} \times d_{\mathcal{Q}}}$: Parameters matrix of linear mapping from original embedding space $\mathcal{X}$ to query space $\mathcal{Q}$.
- $\boldsymbol{W}_{\mathrm{K}} \in \mathbb{R}^{d_{\mathcal{X}} \times d_{\mathcal{K}}}$: Parameters matrix of linear mapping from original embedding space $\mathcal{X}$ to key space $\mathcal{K}$.
- $\boldsymbol{W}_{\mathrm{V}} \in \mathbb{R}^{d_{\mathcal{X}} \times d_{\mathcal{V}}}$: Parameters matrix of linear mapping from original embedding space $\mathcal{X}$ to value space $\mathcal{V}$.

and $\boldsymbol{z}^{(i)} \in \boldsymbol{Z}$ is the encoding vector of i^{th} term containing with the long-distance dependencies information of whole sequence.

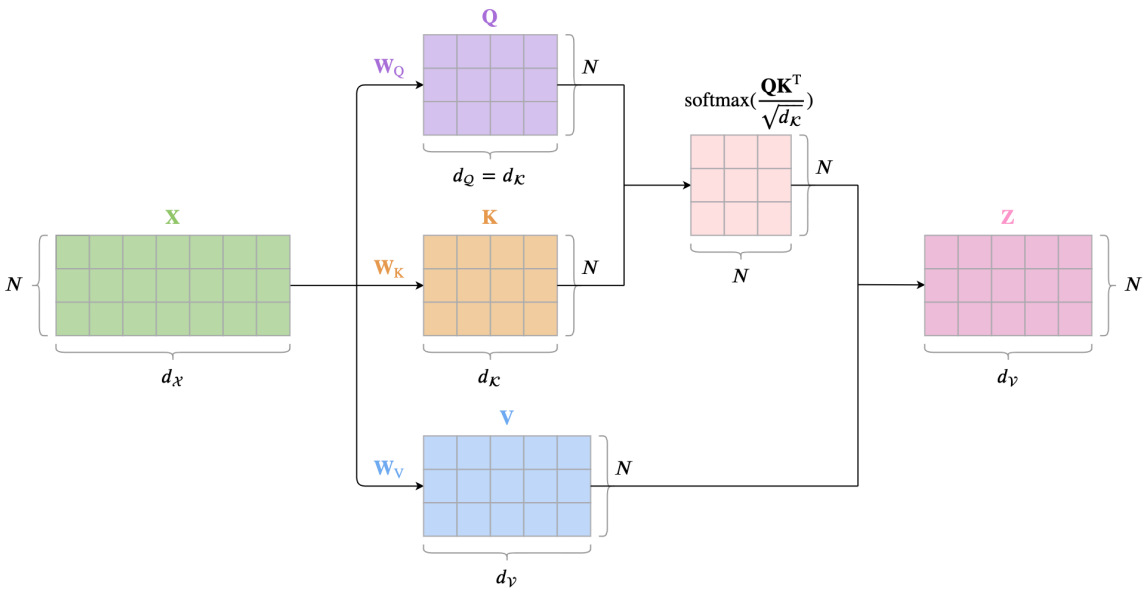


Figure 21: Single-head self-attention.

Multi-Head Attentions. Multi-head attention mechanism improves the performance of the attention layer in two ways:

- Expand the models ability to focus on different positions.
- Give the attention layer multiple "representation subspaces".

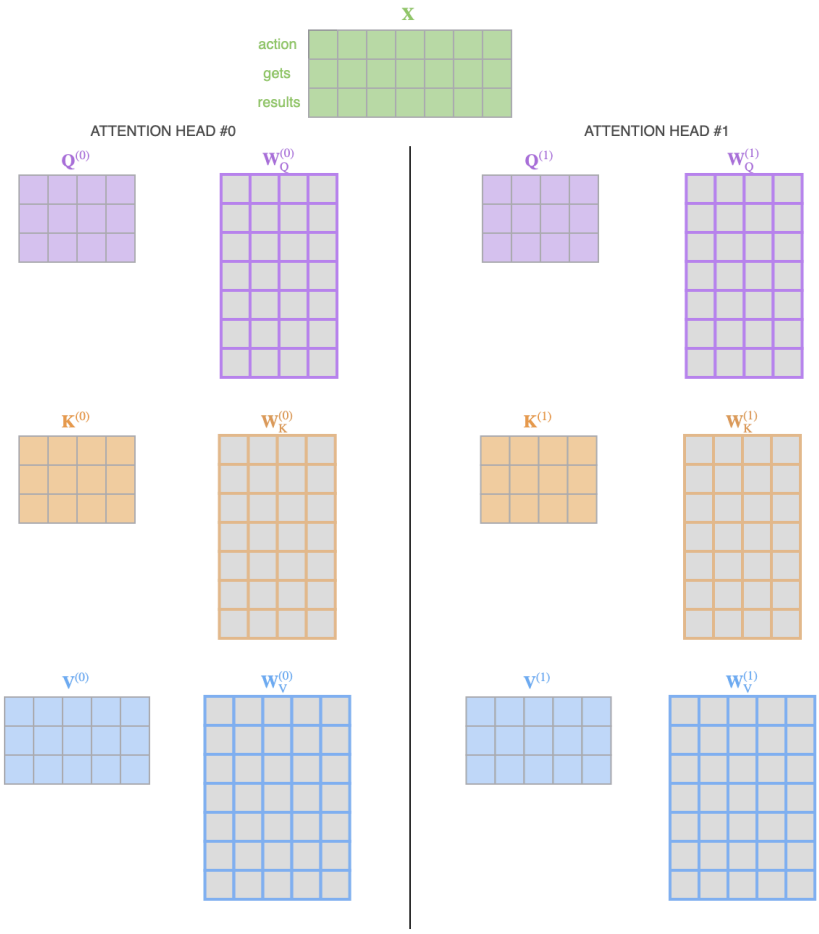


Figure 22: With multi-headed attention, we maintain separate $\boldsymbol{W}_{\mathrm{Q}}/\boldsymbol{W}_{\mathrm{K}}/\boldsymbol{W}_{\mathrm{V}}$ weight matrices for each head resulting in different $\boldsymbol{Q}/\boldsymbol{K}/\boldsymbol{V}$ matrices. As we did before, we multiply $\boldsymbol{X}$ by the different $\boldsymbol{W}_{\mathrm{Q}}/\boldsymbol{W}_{\mathrm{K}}/\boldsymbol{W}_{\mathrm{V}}$ matrices to produce different $\boldsymbol{Q}/\boldsymbol{K}/\boldsymbol{V}$ matrices.

If we do the same self-attention calculation we outlined above, just eight different times with different weight matrices, we end up with eight different $\boldsymbol{Z}$ matrices.

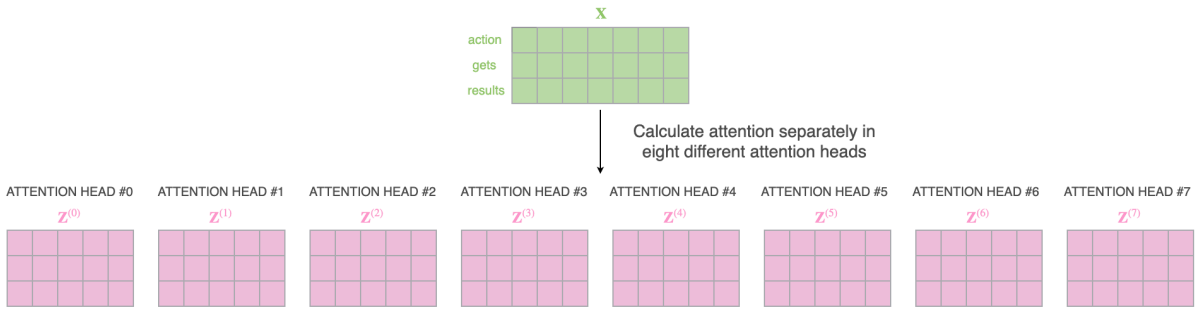


Figure 23: With multi-headed attention, we calculate eight $\boldsymbol{Z}$ separately.

This leaves us with a bit of a challenge. The feed-forward layer is not expecting eight matrices - its expecting a single matrix (a vector for each term). So we need a way to condense these eight down into a single matrix. How do we do that? Transformer model[11] concat the matrices then multiple them by an additional weights matrix $\boldsymbol{W}_{\mathrm{O}}$.

1. Concatenate all the attention heads $\boldsymbol{Z}^{(h)} \in \mathbb{R}^{N \times d_{\mathcal{V}}}$ (Note $\boldsymbol{Z}^{(h)} \in \mathbb{R}^{N \times d_{\mathcal{V}}}$ here is not $\boldsymbol{z}^{(i)} \in \mathbb{R}^{d_{\mathcal{V}}}$ above).

$$\bar{\boldsymbol{Z}} = \boldsymbol{Z}^{(1)} \oplus \cdots \oplus \boldsymbol{Z}^{(H)} \in \mathbb{R}^{N \times H d_{\mathcal{V}}}$$

where $\oplus$ means the concatenation operation of vectors.

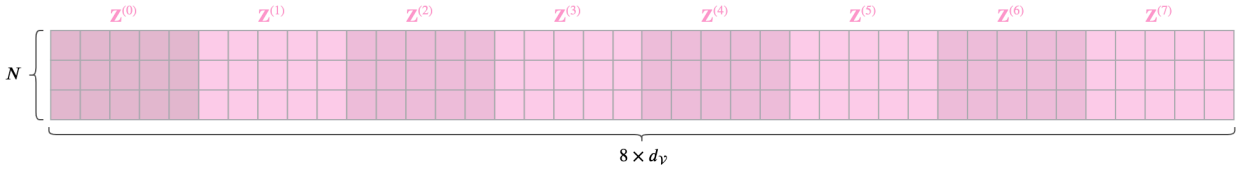


Figure 24: Concatenate multi attention heads $\boldsymbol{Z}^{(h)}, h \in \{1, \cdots, H\}$.

2. Multiply with a weight matrix $\boldsymbol{W}_{\mathrm{O}} \in \mathbb{R}^{H d_{\mathcal{V}} \times d_{\mathcal{Z}}}$ that was trained jointly with the model.

$$\boldsymbol{Z} = \bar{\boldsymbol{Z}} \boldsymbol{W}_{\mathrm{O}} \in \mathbb{R}^{N \times d_{\mathcal{Z}}}$$

The result matrix $\boldsymbol{Z} \in \mathbb{R}^{N \times d_{\mathcal{Z}}}$ captures information from all the attention heads. Then we can send it forward to the FFNN.

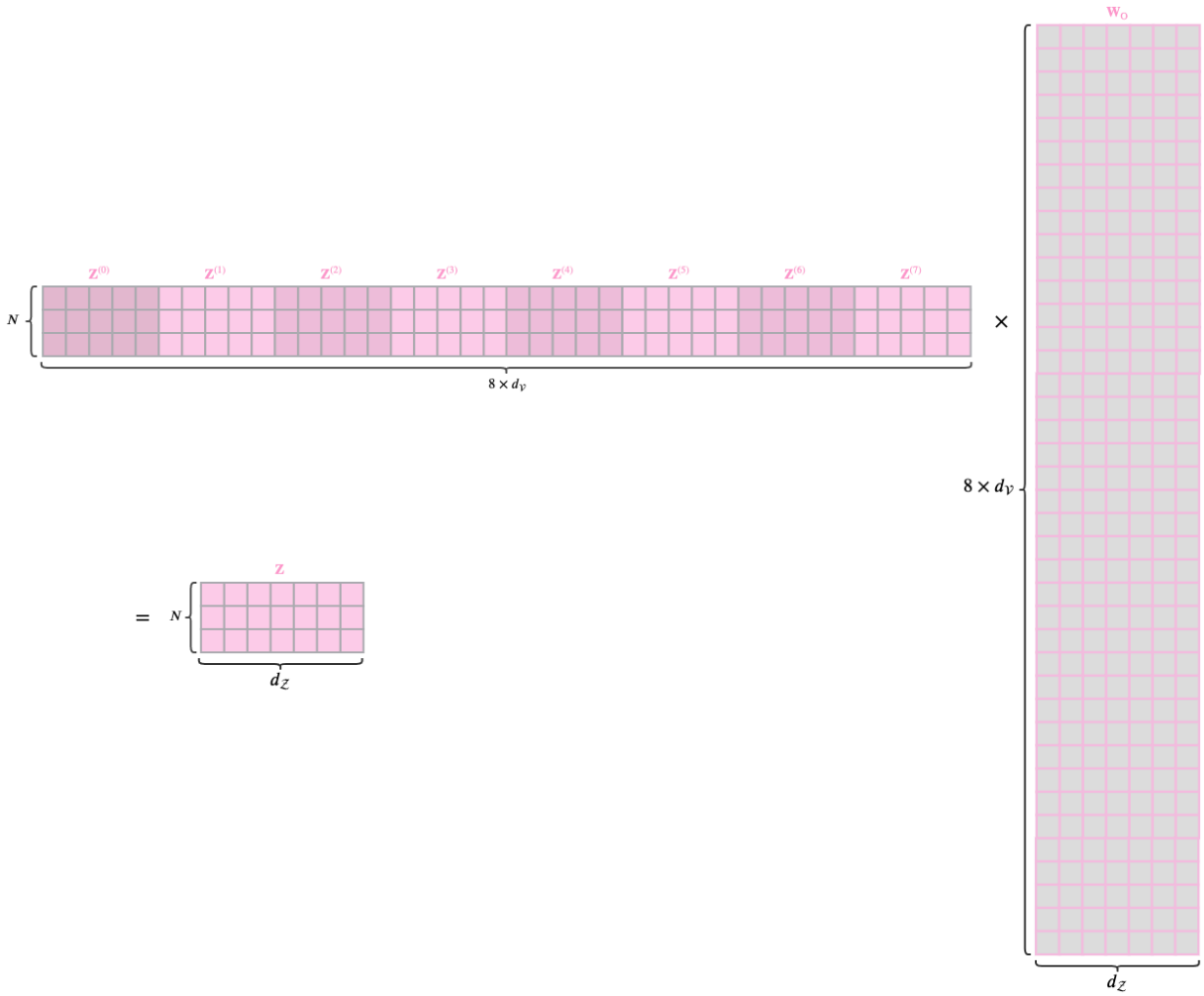


Figure 25: Trainable parameter matrix $\boldsymbol{W}_{\mathrm{O}} \in \mathbb{R}^{H d_{\mathcal{V}} \times d_{\mathcal{Z}}}$ and the result matrix $\boldsymbol{Z} \in \mathbb{R}^{N \times d_{\mathcal{Z}}}$. Note always $d_{\mathcal{Z}} = d_{\mathcal{X}}$ for adding residual connection operation $\boldsymbol{X} + \boldsymbol{Z}$.

Its quite a handful of matrices in multi-head self-attention. Let me try to put them all in one visual so we can look at them in one place.

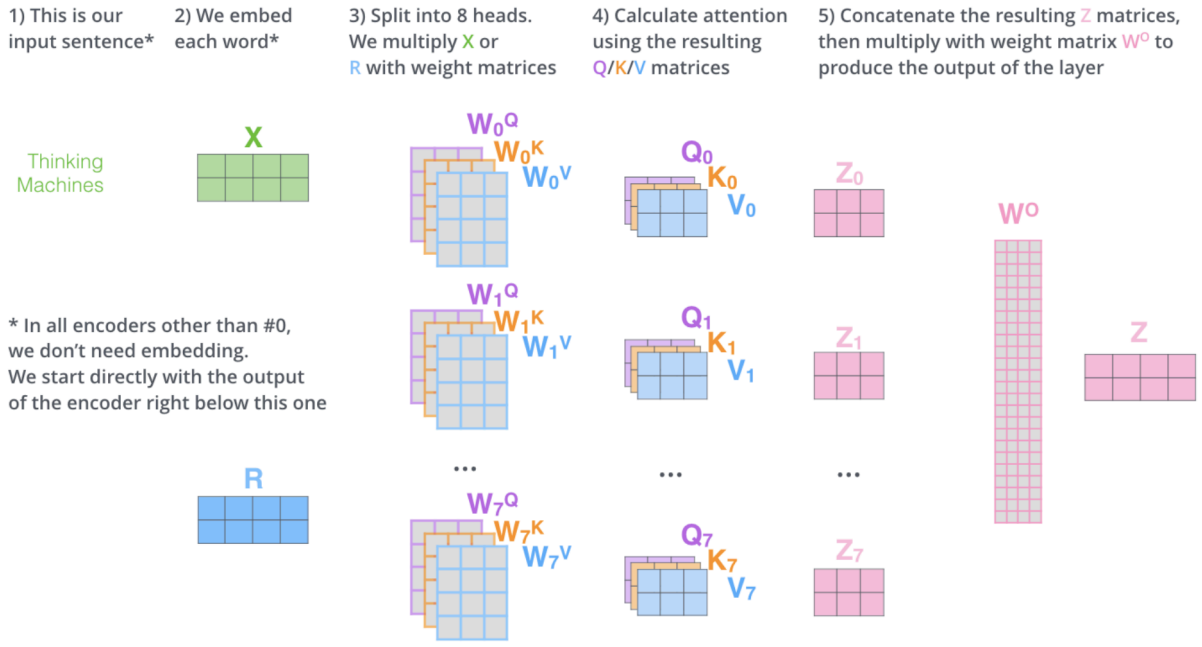


Figure 26: Multi-head self-attentions in one visual. The input embeddings can be either $\boldsymbol{X} \in \mathbb{R}^{N \times d_{\mathcal{X}}}$ (first layer) or $\boldsymbol{R}^{N \times d_{\mathcal{X}}}$ (start from second layer). Source: http://jalammar.github.io/illustrated-transformer/.

Residual Connection & Layer Normalization. After multi-head attention module, we get an encoded matrix $\boldsymbol{Z} \in \mathbb{R}^{N \times d_{\mathcal{Z}}}$, and each row corresponds to an encoded vector $\boldsymbol{z}^{(i)} \in \mathbb{R}^{d_{\mathcal{Z}}}$ containing dependencies on sequence context terms (e.g., words). But in order to alleviate the problem of vanishing gradient or exploding gradient in actual gradient-based optimization, we add a residual connection[40] module in each encoder.

$$\boldsymbol{X} + \boldsymbol{Z} \in \mathbb{R}^{N \times d_{\mathcal{X}}}$$

where $d_{\mathcal{X}} = d_{\mathcal{Z}}$.

Also we want to learn differences among encoding vectors at direction, not meaningless differences of numerical size, therefore we impose a layer normalization[41] operation immediately after residual connection.

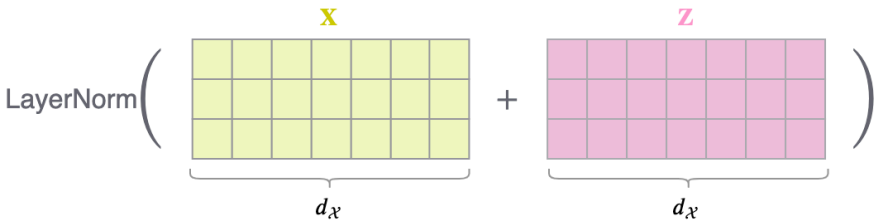


Figure 27: Residual connection & layer normalization after multi-head attention. $\tilde{\boldsymbol{Z}} = \text{LayerNorm}(\boldsymbol{X} + \boldsymbol{Z}) \in \mathbb{R}^{N \times d_{\mathcal{X}}}$.

Feed Forward Neural Network. The linear transformations are the same and separate across different positions, indeed $(\boldsymbol{x}^{(i)} + \boldsymbol{z}^{(i)}) \in \mathbb{R}^{d_{\mathcal{X}}}$ in a sequence:

$$\text{FFN}(\tilde{\boldsymbol{Z}}) = \max(0, \tilde{\boldsymbol{Z}}\boldsymbol{W}_1 + \boldsymbol{b}_1)\boldsymbol{W}_2 + \boldsymbol{b}_2 \in \mathbb{R}^{N \times d_{\mathcal{X}}}$$

where $\boldsymbol{W}_1 \in \mathbb{R}^{d_{\mathcal{X}} \times d_{\text{ff}}}$, $\boldsymbol{W}_2 \in \mathbb{R}^{d_{\text{ff}} \times d_{\mathcal{X}}}$, $\boldsymbol{b}_1 \in \mathbb{R}^{d_{\text{ff}}}$ and $\boldsymbol{b}_2 \in \mathbb{R}^{d_{\mathcal{X}}}$ are trainable parameters in FFN module, and $\tilde{\boldsymbol{Z}} \in \mathbb{R}^{N \times d_{\mathcal{X}}}$ is the output after residual connection & layer normalization, namely:

$$\tilde{\boldsymbol{Z}} = \text{LayerNorm}(\boldsymbol{X} + \boldsymbol{Z}) \in \mathbb{R}^{N \times d_{\mathcal{X}}}$$

Encoder. The above describes all basic components in each encoder. Each basic encoder layer contains all components as follows:

- Multi-head attention (consists of multiple self-attention modules)
- Residual connection & layer normalization after multi-head attention
- Feed forward neural network
- Residual connection & layer normalization after feed forward neural network

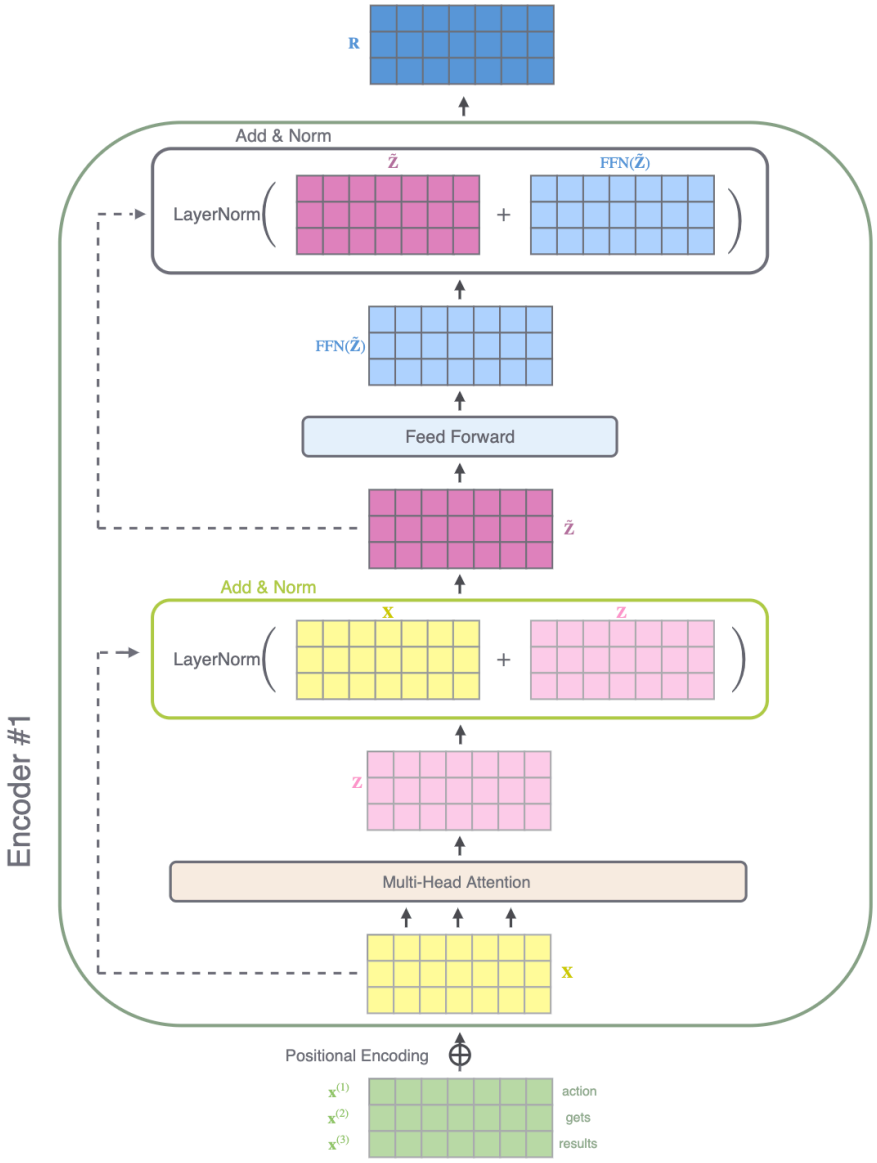


Figure 28: Visualize the first encoder layer of Transformer.

In each encoder module we have:

- Input tensor: $\boldsymbol{X} \in \mathbb{R}^{N \times d_{\mathcal{X}}}$ or $\boldsymbol{R} \in \mathbb{R}^{N \times d_{\mathcal{X}}}$
- Output tensor: $\boldsymbol{R} \in \mathbb{R}^{N \times d_{\mathcal{X}}}$
- Trainable parameters:
  - Multi-head attention parameters:

  - $\boldsymbol{W}_{\mathrm{Q}}^{(h)} \in \mathbb{R}^{d_{\mathcal{X}} \times d_{\mathcal{Q}}},\ \forall h \in \{1, \cdots, H\}$
  - $\boldsymbol{W}_{\mathrm{K}}^{(h)} \in \mathbb{R}^{d_{\mathcal{X}} \times d_{\mathcal{K}}},\ \forall h \in \{1, \cdots, H\} \quad (d_{\mathcal{K}} = d_{\mathcal{Q}})$
  - $\boldsymbol{W}_{\mathrm{V}}^{(h)} \in \mathbb{R}^{d_{\mathcal{X}} \times d_{\mathcal{V}}},\ \forall h \in \{1, \cdots, H\}$
  - $\boldsymbol{W}_{\mathrm{O}} \in \mathbb{R}^{d_{\mathcal{V}} \times d_{\mathcal{X}}}$
- Feed forward network parameters:
  - $\boldsymbol{W}_{\mathrm{ffn}-1} \in \mathbb{R}^{d_{\mathcal{X}} \times d_{\mathrm{ff}}}$
  - $\boldsymbol{W}_{\mathrm{ffn}-2} \in \mathbb{R}^{d_{\mathrm{ff}} \times d_{\mathcal{X}}}$
  - $\boldsymbol{b}_{\mathrm{ffn}-1} \in \mathbb{R}^{d_{\mathrm{ff}}}$
  - $\boldsymbol{b}_{\mathrm{ffn}-2} \in \mathbb{R}^{d_{\mathcal{X}}}$
- Layer norm parameters

Masked Multi-Head Attention. In Transformer model, there exists two kinds of mask:

- Padding Mask
- Sequence Mask

here is the sequence mask, in paper [11], authors modified the self-attention sub-layer in the decoder stack to prevent positions from attending to subsequent positions. This masking, combined with fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i. Mathematically explain as follows:

$$\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}} = \begin{pmatrix} \frac{\boldsymbol{q}^{(1)} \cdot \boldsymbol{k}^{(1)}}{\sqrt{d_{\mathcal{K}}}} & \cdots & \frac{\boldsymbol{q}^{(1)} \cdot \boldsymbol{k}^{(j)}}{\sqrt{d_{\mathcal{K}}}} & \cdots & \frac{\boldsymbol{q}^{(1)} \cdot \boldsymbol{k}^{(N)}}{\sqrt{d_{\mathcal{K}}}} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{\boldsymbol{q}^{(i)} \cdot \boldsymbol{k}^{(1)}}{\sqrt{d_{\mathcal{K}}}} & \cdots & \frac{\boldsymbol{q}^{(i)} \cdot \boldsymbol{k}^{(j)}}{\sqrt{d_{\mathcal{K}}}} & \cdots & \frac{\boldsymbol{q}^{(i)} \cdot \boldsymbol{k}^{(N)}}{\sqrt{d_{\mathcal{K}}}} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{\boldsymbol{q}^{(N)} \cdot \boldsymbol{k}^{(1)}}{\sqrt{d_{\mathcal{K}}}} & \cdots & \frac{\boldsymbol{q}^{(N)} \cdot \boldsymbol{k}^{(j)}}{\sqrt{d_{\mathcal{K}}}} & \cdots & \frac{\boldsymbol{q}^{(N)} \cdot \boldsymbol{k}^{(N)}}{\sqrt{d_{\mathcal{K}}}} \end{pmatrix} \in \mathbb{R}^{N \times N}$$

where $\frac{\boldsymbol{q}^{(i)} \cdot \boldsymbol{k}^{(j)}}{\sqrt{d_{\mathcal{K}}}}$ means the importance (not normalized) of j-th token on i-th token, compared among all tokens (in the same row of matrix) when encoding i-th token. In decoder, when encode i-th token, we can only use the information of tokens before i-th position, we can't see the future (i.e., can't use information of tokens at $\{j \mid j > i\}$ positions) , namely we need a mask matrix:

$$\boldsymbol{M} = \begin{pmatrix} 0 & -\inf & \cdots & -\inf \\ 0 & -\inf & \cdots & -\inf \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 \end{pmatrix} \in \mathbb{R}^{N \times N}$$

where $\boldsymbol{M}$ is a upper triangular matrix, and $\mathrm{M}_{ij} \in \{0, -\inf\}$ means whether i-th token need to mask out the influence of the j-th token on it, if $\mathrm{M}_{ij} = -\inf$, mask out!

Then we mask the unnormalized attention matrix $\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}}$ with a very negative number (e.g., -10^9), so that after softmax operation later, it can be a very tiny fraction (e.g., $-10^9 \to 0.00012$)

$$\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}} + \boldsymbol{M} = \begin{pmatrix} \frac{\boldsymbol{q}^{(1)} \cdot \boldsymbol{k}^{(1)}}{\sqrt{d_{\mathcal{K}}}} & \cdots & -\inf & \cdots & -\inf \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{\boldsymbol{q}^{(i)} \cdot \boldsymbol{k}^{(1)}}{\sqrt{d_{\mathcal{K}}}} & \cdots & \frac{\boldsymbol{q}^{(i)} \cdot \boldsymbol{k}^{(i)}}{\sqrt{d_{\mathcal{K}}}} & \cdots & -\inf \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{\boldsymbol{q}^{(N)} \cdot \boldsymbol{k}^{(1)}}{\sqrt{d_{\mathcal{K}}}} & \cdots & \frac{\boldsymbol{q}^{(N)} \cdot \boldsymbol{k}^{(j)}}{\sqrt{d_{\mathcal{K}}}} & \cdots & \frac{\boldsymbol{q}^{(N)} \cdot \boldsymbol{k}^{(N)}}{\sqrt{d_{\mathcal{K}}}} \end{pmatrix} \in \mathbb{R}^{N \times N}$$

And finally, we impose softmax operation along row:

$$\mathrm{softmax}(\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}} + \boldsymbol{M}) \in \mathbb{R}^{N \times N}$$

where $\mathrm{softmax}(\cdot)$ is an operation along row.

Encoder-Decoder Attention. Encoder-decoder attention also belongs to multi-head attention, but the only difference between a decoder and an encoder is that

- $\boldsymbol{Q}^{(h)},\ \forall h \in \{1, \cdots H\}$: from previous decoder layer.
- $\boldsymbol{K}^{(h)},\ \forall h \in \{1, \cdots H\}$: output of the last encoder.
- $\boldsymbol{V}^{(h)},\ \forall h \in \{1, \cdots H\}$: output of the last encoder.

the reason why $\boldsymbol{K}^{(h)}$ and $\boldsymbol{V}^{(h)}$ come from source domain and $\boldsymbol{Q}^{(h)}$ comes from target domain is that we need the information interaction between original domain and target domain.

Therefore, we can get the output of a specific head self-attention module as follows:

$$\boldsymbol{Z}^{(h)} = \text{softmax}(\frac{\boldsymbol{Q}^{(h)}(\boldsymbol{K}^{(h)})^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}} + \boldsymbol{M}^{(h)})\boldsymbol{V}^{(h)},\ \forall h \in \{1, \cdots H\}$$

always we have

$$\boldsymbol{M}^{(1)} = \cdots = \boldsymbol{M}^{(H)} = \boldsymbol{M}$$

and then we can get the output of masked multi-head attention module:

$$\boldsymbol{Z} = (\boldsymbol{Z}^{(1)} \oplus \cdots \oplus \boldsymbol{Z}^{(H)})\boldsymbol{W}_{\mathrm{O}} \in \mathbb{R}^{N \times d_{\mathcal{X}}}$$

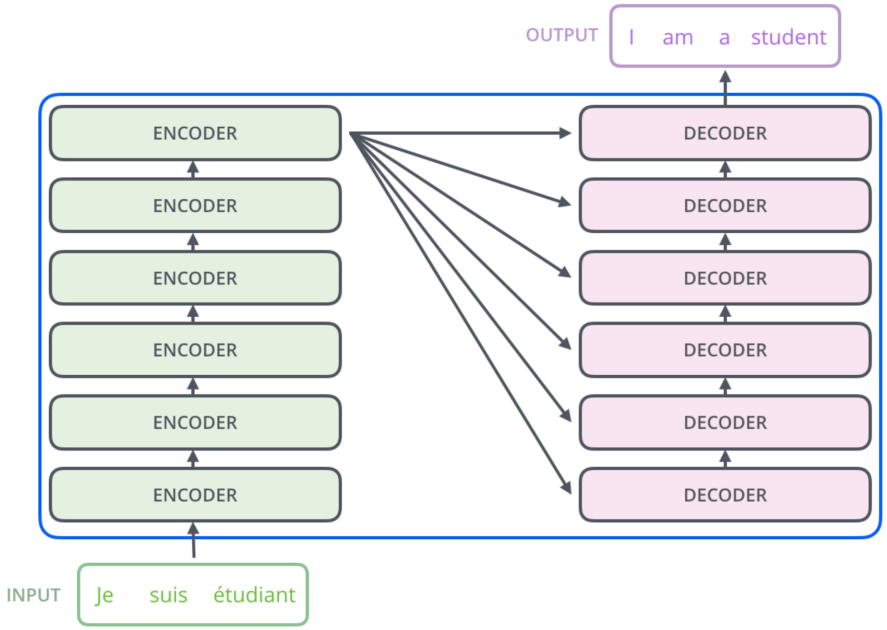


Figure 29: Connections between the last encoder and all decoders. Source: http://jalammar.github.io/illustrated-transformer/.

Decoder. Each decoder stacks following components from bottom to top (See Figure 20):

- Masked Multi-Head Attention (Masked Multi-Head Self-Attention)
- Add & Norm
- Multi-Head Attention (Multi-Head Encoder-Decoder Attention)
- Add & Norm
- Feed Forward
- Add & Norm

Note that, in decoder, we use two types of multi-head attention, and the Transformer uses self attention (scaled dot-product attention) in three different ways[11]:

- In "encoder-decoder self-attention" layer, the queries $\boldsymbol{Q}^{(h)}$ come from the previous decoder layer, and the memory keys $\boldsymbol{K}^{(h)}$ and values $\boldsymbol{V}^{(h)}$ come from the output of the last encoder. This allows every position in the decoder to attend over all positions in the input sequence. This mimics the typical encoder-decoder attention mechanisms in sequence-to-sequence models.
- In "encoder self-attention" layer, queries $\boldsymbol{Q}^{(h)}$, keys $\boldsymbol{K}^{(h)}$ and values $\boldsymbol{V}^{(h)}$ come from the same place, in this case, the output of the previous layer in the encoder. Each position in the encoder can attend to all positions in the previous layer of the encoder.
- In "decoder self-attention" layer, it allows each position (i.e., query position i) in the decoder to attend to all positions (i.e., key positions j) in the decoder up to and including that position (i.e., attend to $\{j \,|\, j \leq i\}$, mask out $\{j \,|\, j > i\}$). We need to prevent leftward information ow (i.e., "see the future") in the decoder in the decoder to preserve the auto-regressive property. We implement this inside of scaled dot-product attention by masking out (setting to $-\infty$) all values in the input of the softmax which correspond to illegal connections. See Figure 7.

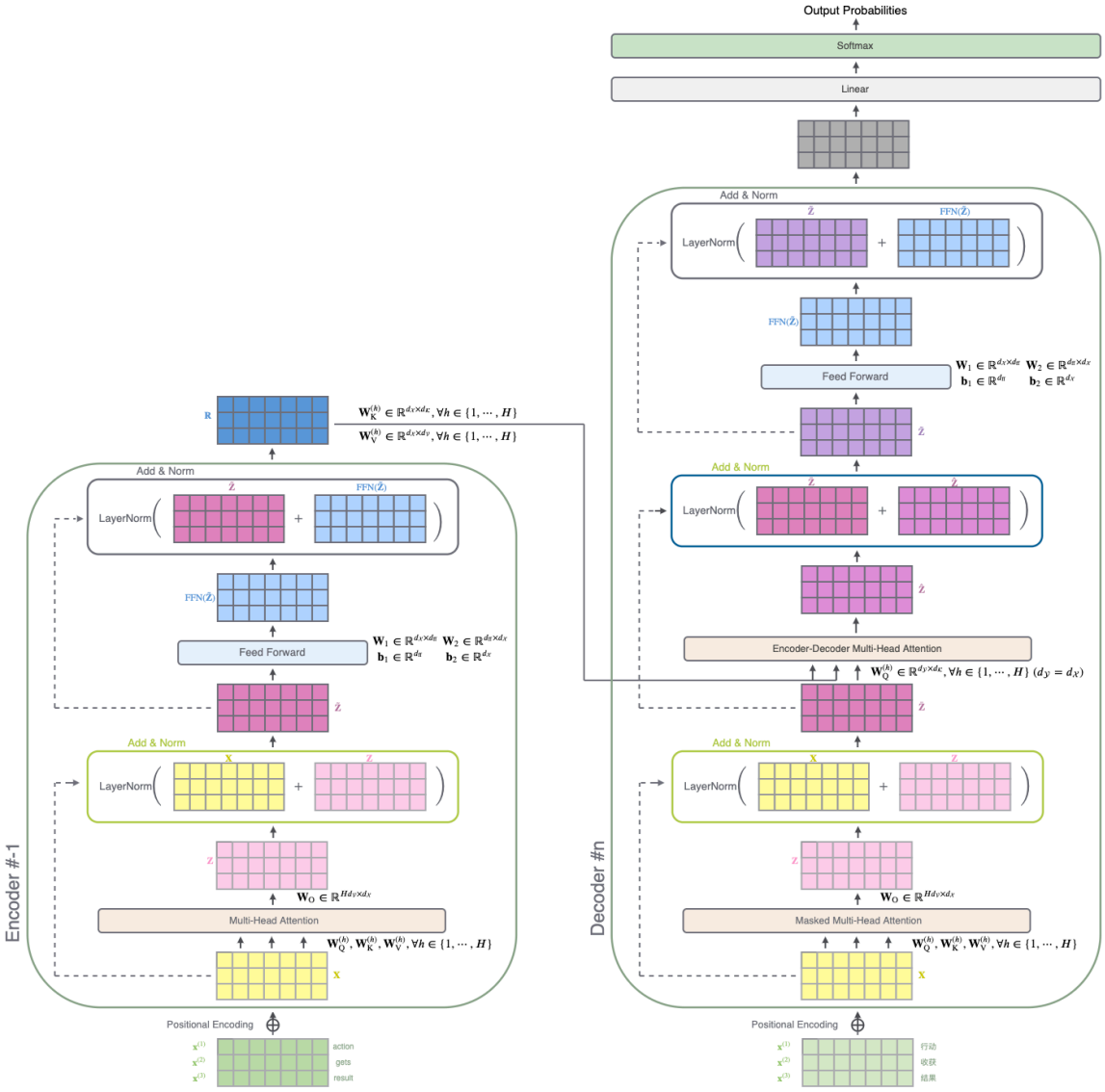


Figure 30: The expanded Transformer - model architecture. Note that $d_{\mathcal{X}} = d_{\mathcal{Y}}$ and $N_{\mathcal{X}} = N_{\mathcal{Y}}$ must be met.

Positional Encoding. **(Not Accomplished)** One thing thats missing from self-attention model as we have described it so far is a way to account for the order of the terms (e.g., words) in the input sequence $\boldsymbol{X}$. Namely, self-attention mechanism only account for the long-distance dependences on the content of term (e.g., word embedding), but does not include the information about the relative position or absolute position. Because each encoding vector corresponding i^{th} term in sequence $\boldsymbol{z}^{(i)}$ is:

$$\boldsymbol{z}^{(i)} = \sum_{j=1}^{N} \frac{\exp(\frac{\boldsymbol{q}^{(i)} \cdot \boldsymbol{k}^{(j)}}{\sqrt{d_{\mathcal{K}}}})}{\sum_{l=1}^{N} \exp(\frac{\boldsymbol{q}^{(1)} \cdot \boldsymbol{k}^{(\ell)}}{\sqrt{d_{\mathcal{K}}}})} \boldsymbol{v}^{(j)} \in \mathbb{R}^{d_{\mathcal{V}}}$$

it's only the weighted average sum of all value vectors of all terms, but not contain the positional information of a sequence.

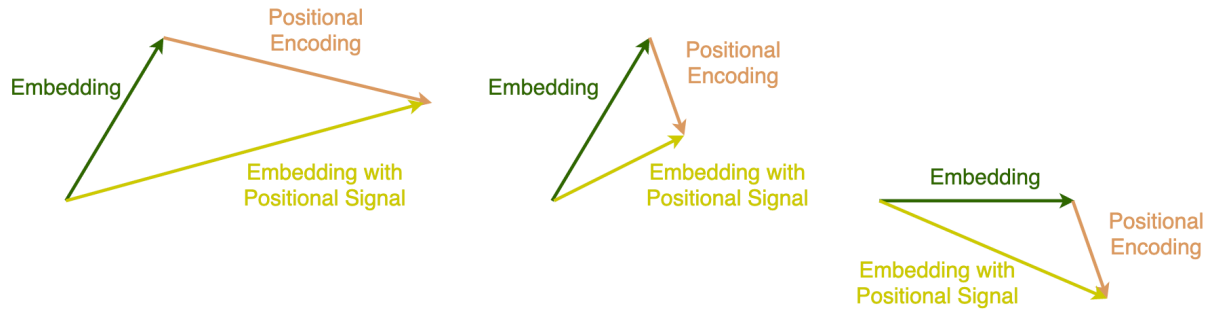


Figure 31: Embedding with positional encoding. Same embedding add with different positional encodings generate different result embeddings. Different embeddings add with same positional encoding generate different result embeddings. Note the positional encodings have the same dimension $d_{\mathcal{X}}$ as the embeddings, so that the two can be summed.

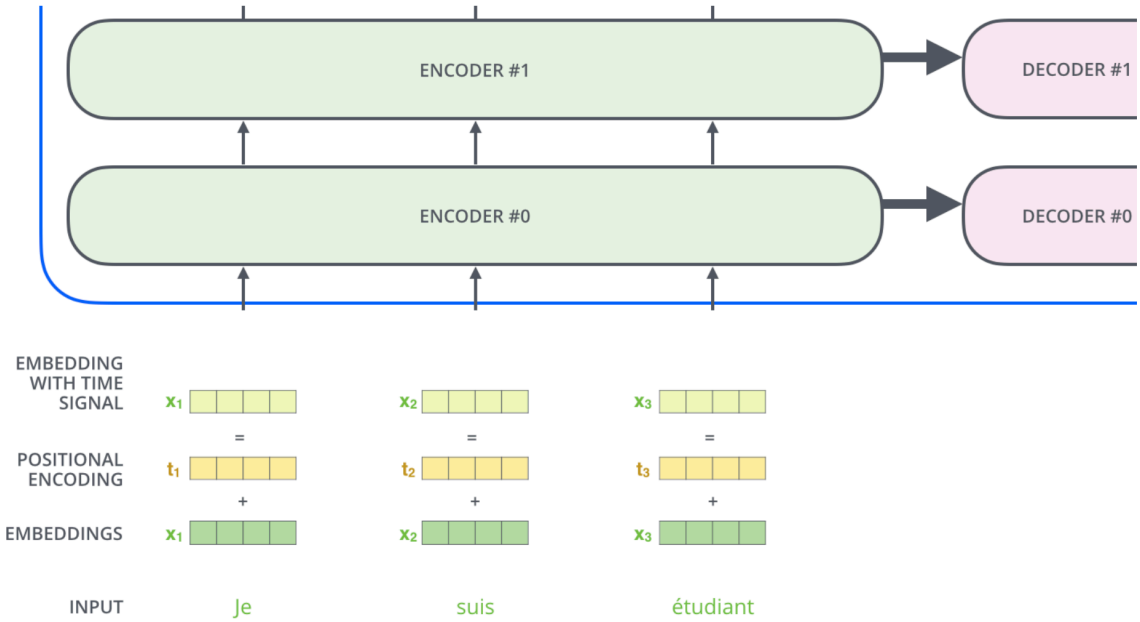


Figure 32: To give the model a sense of the order of the words, we add positional encoding vectors – the values of which follow a specific pattern. Source: http://jalammar.github.io/illustrated-transformer/.

To address this, the transformer adds a vector to each input embedding. These vectors follow a specific pattern that the model learns, which helps it determine the position of each word, or the distance between different words in the sequence. The intuition here is that adding these values to the embeddings provides meaningful distances between the embedding vectors in original embedding space $\mathcal{X}$ before once they're projected into $\boldsymbol{q}/\boldsymbol{k}/\boldsymbol{v}$ vectors and during dot-product attention.

Note the positional encodings have the same dimension $d_{\mathcal{X}}$ as the embeddings, so that the two can be summed.

In paper [11], they use sine and cosine functions of different frequencies:

$$\text{PE}(\text{pos}, 2i) = \sin(\text{pos}/10000^{\frac{2i}{d_{\mathcal{X}}}})$$
$$\text{PE}(\text{pos}, 2i+1) = \cos(\text{pos}/10000^{\frac{2i}{d_{\mathcal{X}}}})$$

where pos is the position of term in sequence, i is the dimension.

If we assumed the embedding has a dimensionality of 4, the actual positional encodings would look like this:

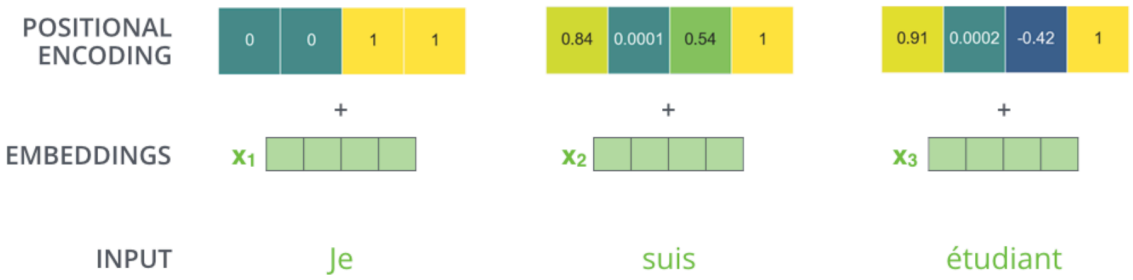


Figure 33: A real example of positional encoding with a toy embedding size of 4. Source: http://jalammar.github.io/illustrated-transformer/.

Training.

- Teacher Forcing: Using ground-truth $\text{Y} = [y^{(1)}, \cdots, y^{(N)}]$ as the input sequence of decoder part.
- Scheduled Sampling: Sometimes ground-truth $\text{Y} = [y^{(1)}, \cdots, y^{(N)}]$ as the input sequence of decoder part, sometimes predicted sequence until now $\text{Y} = [\hat{y}^{(1)}, \cdots, \hat{y}^{(N)}]$ as the input sequence of decoder part.
- Greedy Search $\rightarrow$ Beam Search: Not always select the token with maximum probability in softmax of decoder part.

Inference. Suppose we have a trained Transformer network, when making predictions, the steps are as follows:

1. In encoder part, input the whole source embedding sequence $\boldsymbol{X} = [\boldsymbol{x}^{(1)}, \cdots, \boldsymbol{x}^{(N)}]^{\text{T}} \in \mathbb{R}^{N \times d_{\mathcal{X}}}$ to be translated; In decoder part, input the first token (i.e., $y^{(1)} =$ </s>) embedding sequence $\boldsymbol{Y} = [\boldsymbol{y}^{(1)}]^{\text{T}} \in \mathbb{R}^{1 \times d_{\mathcal{X}}}$, get the output of decoder at the 1-th output position $\hat{y}^{(1)} \in \{1, \cdots, N_{\text{word}}\}$.
2. For making the translation of i-th position in target sequence, we get the whole source embedding sequence $\boldsymbol{X} = [\boldsymbol{x}^{(1)}, \cdots, \boldsymbol{x}^{(N)}]^{\text{T}} \in \mathbb{R}^{N \times d_{\mathcal{X}}}$ at encoder part. At decoder part, we get the input sequence until now $\text{Y} = [y^{(1)}, \cdots, y^{(i-1)}]$, corresponds with target embedding sequence until now $\text{Y} \rightarrow \boldsymbol{Y} = [\boldsymbol{y}^{(1)}, \cdots, \boldsymbol{y}^{(i-1)}]^{\text{T}} \in \mathbb{R}^{(i-1) \times d_{\mathcal{X}}}$, then we get the output sequence $\hat{\boldsymbol{Y}} = [\hat{y}^{(1)}, \cdots, \hat{y}^{(i)}]$.
3. If we get the output $\hat{\boldsymbol{y}}^{(i)} =$ </eos>, end the inference process.

21.1.4 Cognition

Opinions.

- Different mask prior information matrix $\boldsymbol{M}^{(h)}$ for different heads in multi-head attention module, just like the randomness of dropout.

Reading Materials.

- Attention Is All You Need (Vaswani et al., 2017)[11].
- Jianshu: Twelve Hours teaches you intuitively the Position-Encoding.
- Timo Denk's Blog: Linear Relationships in the Transformers Positional Encoding.
- Jay Alammar: The Illustrated Transformer.
- harvardnlp: The Annotated Transformer.

21.2 Transformer-XL

21.2.1 Definition

21.2.2 Cognition

Reading Materials.

- Transformer-XL: Attentive Language Models beyond a Fixed-Length Context (Dai et al., 2019)[42].

21.3 Reformers (Not Accomplished)

21.3.1 Motivation

- Training Transformer models[11] can be prohibitively costly, especially on long sequences[43].
- Cannot be fine-turned on a single GPU[43].

Consider the following calculation:

- Per-layer memory: 0.5B parameters used in the largest reported Transformer layer (i.e., multi-head attention & feed forward network) account for 2GB of memory, namely $\left\{\boldsymbol{W}_{\mathrm{Q}}^{(h)}, \boldsymbol{W}_{\mathrm{K}}^{(h)}, \boldsymbol{W}_{\mathrm{V}}^{(h)} \,|\, h \in \{1, \cdots, H\}\right\} \cup \left\{\boldsymbol{W}_{\mathrm{O}}\right\} \cup \left\{\boldsymbol{W}_1, \boldsymbol{b}_1, \boldsymbol{W}_2, \boldsymbol{b}_2\right\}$.
- Mini-batch dataset memory: mini-batch size $B = 8$ and $N = 64$K tokens in a sequence with embedding size $d_{\mathcal{X}} = 1024$ requiring another $8 \times 64\mathrm{K} \times 1\mathrm{K} \times 32\mathrm{bits} = 2^{34}\mathrm{bits} = 2^{31}\mathrm{Bytes} = 2^{21}\mathrm{KB} = 2^{11}\mathrm{MB} = 2\mathrm{GB}$ of memory, namely $\boldsymbol{X}$.

The above estimate includes only per-layer memory and input activations (i.e., mini-batch source training dataset) cost and does not take into account the following major sources of memory usage in the Transformer:

- Memory in a model with L layers is L-times larger than in a single-layer model due to the fact that activations $\boldsymbol{R}$ in Figure 28 need to be stored for back-propagation.
- Since the depth d_{ff} of intermediate feed-forward layers is often much larger than the depth $d_{\mathcal{X}}$ of attention activations $\tilde{\boldsymbol{Z}}$ in Figure 28, it accounts for a large fraction of memory use.
- Attention on sequences $\mathrm{softmax}(\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}})$ of length N is $\mathrm{O}(N^2)$ in both computational and memory complexity, so even for a single sequence of 64K tokens can exhaust accelerator memory.

In other words, the original Transformer is too memory-cost and computation-cost to be reduced and optimized.

21.3.2 Contribution

Introducing the Reformer model which solves these problems using the following techniques:

- Reversible layers, first introduced in (Gomez et al., 2017)[44], enable storing only a single copy of activations in the whole model, so the L (number of stacking of Transformer layers) factor disappears.
- Splitting activations inside feed-forward layers and processing them in chunks removes the d_{ff} factor and saves memory inside feed-forward layers.
- Approximate attention computation (i.e., $\mathrm{softmax}(\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}})$) based on locality-sensitive hashing (LSH) replaces the $\mathrm{O}(N^2)$ factor in each attention layers with $\mathrm{O}(N \log N)$ and so allows operating on long sequences.

21.3.3 Definition

Locality-Sensitive Hashing Attention.

Dot-product attention. In practice, the attention function on a set of queries is computed simultaneously, packed together into a matrix $\boldsymbol{Q} \in \mathbb{R}^{N \times d_{\mathcal{K}}}$ (where N is the length of sequences, not batch size here, same as follows). Assuming the keys and values are also packed together into matrices $\boldsymbol{K} \in \mathbb{R}^{N \times d_{\mathcal{K}}}$ and $\boldsymbol{V} \in \mathbb{R}^{N \times d_{\mathcal{V}}}$, the matrix outputs with attention mechanism (see 20.1) is defined as:

$$\mathrm{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \mathrm{softmax}(\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}})\boldsymbol{V}$$

$$
= \begin{pmatrix}
\frac{\exp(\frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(1)}}{\sqrt{d_\mathcal{K}}})}{\sum_{\ell=1}^{N}\exp(\frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_\mathcal{K}}})} & \cdots & \frac{\exp(\frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(j)}}{\sqrt{d_\mathcal{K}}})}{\sum_{\ell=1}^{N}\exp(\frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_\mathcal{K}}})} & \cdots & \frac{\exp(\frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(N)}}{\sqrt{d_\mathcal{K}}})}{\sum_{\ell=1}^{N}\exp(\frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_\mathcal{K}}})} \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
\frac{\exp(\frac{\boldsymbol{q}^{(i)}\cdot\boldsymbol{k}^{(1)}}{\sqrt{d_\mathcal{K}}})}{\sum_{\ell=1}^{N}\exp(\frac{\boldsymbol{q}^{(i)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_\mathcal{K}}})} & \cdots & \frac{\exp(\frac{\boldsymbol{q}^{(i)}\cdot\boldsymbol{k}^{(j)}}{\sqrt{d_\mathcal{K}}})}{\sum_{\ell=1}^{N}\exp(\frac{\boldsymbol{q}^{(i)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_\mathcal{K}}})} & \cdots & \frac{\exp(\frac{\boldsymbol{q}^{(i)}\cdot\boldsymbol{k}^{(N)}}{\sqrt{d_\mathcal{K}}})}{\sum_{\ell=1}^{N}\exp(\frac{\boldsymbol{q}^{(i)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_\mathcal{K}}})} \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
\frac{\exp(\frac{\boldsymbol{q}^{(N)}\cdot\boldsymbol{k}^{(1)}}{\sqrt{d_\mathcal{K}}})}{\sum_{\ell=1}^{N}\exp(\frac{\boldsymbol{q}^{(N)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_\mathcal{K}}})} & \cdots & \frac{\exp(\frac{\boldsymbol{q}^{(N)}\cdot\boldsymbol{k}^{(j)}}{\sqrt{d_\mathcal{K}}})}{\sum_{\ell=1}^{N}\exp(\frac{\boldsymbol{q}^{(N)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_\mathcal{K}}})} & \cdots & \frac{\exp(\frac{\boldsymbol{q}^{(N)}\cdot\boldsymbol{k}^{(N)}}{\sqrt{d_\mathcal{K}}})}{\sum_{\ell=1}^{N}\exp(\frac{\boldsymbol{q}^{(N)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_\mathcal{K}}})}
\end{pmatrix}
\begin{pmatrix}
(\boldsymbol{v}^{(1)})^\mathrm{T} \\
\vdots \\
(\boldsymbol{v}^{(j)})^\mathrm{T} \\
\vdots \\
(\boldsymbol{v}^{(N)})^\mathrm{T}
\end{pmatrix}
$$

$$
= \begin{pmatrix}
\sum_{j=1}^{N}\frac{\exp(\frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(j)}}{\sqrt{d_\mathcal{K}}})}{\sum_{\ell=1}^{N}\exp(\frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_\mathcal{K}}})}(\boldsymbol{v}^{(j)})^\mathrm{T} \\
\vdots \\
\sum_{j=1}^{N}\frac{\exp(\frac{\boldsymbol{q}^{(i)}\cdot\boldsymbol{k}^{(j)}}{\sqrt{d_\mathcal{K}}})}{\sum_{\ell=1}^{N}\exp(\frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_\mathcal{K}}})}(\boldsymbol{v}^{(j)})^\mathrm{T} \\
\vdots \\
\sum_{j=1}^{N}\frac{\exp(\frac{\boldsymbol{q}^{(N)}\cdot\boldsymbol{k}^{(j)}}{\sqrt{d_\mathcal{K}}})}{\sum_{\ell=1}^{N}\exp(\frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_\mathcal{K}}})}(\boldsymbol{v}^{(j)})^\mathrm{T}
\end{pmatrix} \in \mathbb{R}^{N\times d_\mathcal{V}}
$$

where softmax$(\cdot)$ is computed along row here.

Memory-efficient attention. To calculate the memory usage of the self-attention mechanism, let us focus on the attention computation from above Attention$(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V})$. Let us assume $\boldsymbol{Q}$, $\boldsymbol{K}$ and $\boldsymbol{V}$ all have the shape $(\text{BatchSize}, N, d_\mathcal{X})$. The main issue is the term $\boldsymbol{Q}\boldsymbol{K}^\mathrm{T}$, which has the shape $(\text{BatchSize}, N, N)$. If we train a model on sequences of length 64K - in this case, even at batch size of 1, this is a 64K $\times$ 64K matrix, which in 32-bit floats will take $2^{16}\times 2^{16}\times 2^5\text{bits} = 2^{16}\times 2^{16}\times 2^2\text{Bytes} = 2^{34}\text{Bytes} = 2^{24}\text{KB} = 2^{14}\text{MB} = 2^4\text{GB} = 16\text{GB}$ of memory.

This is impractical and has hindered the use of the Transformer for long sequences. But it is important to note that the $\boldsymbol{Q}\boldsymbol{K}^\mathrm{T}$ matrix does not need to be fully materialized in memory.

The attention can indeed be computed for each query $\boldsymbol{q}^{(i)} \in \mathbb{R}^{d_\mathcal{K}}$ separately, only calculating as follows once in memory:

$$
\text{softmax}(\frac{(\boldsymbol{q}^{(i)})^\mathrm{T}\boldsymbol{K}^\mathrm{T}}{\sqrt{d_\mathcal{K}}})\boldsymbol{V} = \text{softmax}(\begin{pmatrix}\frac{\boldsymbol{q}^{(i)}\cdot\boldsymbol{k}^{(1)}}{\sqrt{d_\mathcal{K}}} & \cdots & \frac{\boldsymbol{q}^{(i)}\cdot\boldsymbol{k}^{(j)}}{\sqrt{d_\mathcal{K}}} & \cdots & \frac{\boldsymbol{q}^{(i)}\cdot\boldsymbol{k}^{(N)}}{\sqrt{d_\mathcal{K}}}\end{pmatrix})\begin{pmatrix}\boldsymbol{v}^{(1)},\boldsymbol{v}^{(2)},\cdots,\boldsymbol{v}^{(N)}\end{pmatrix}^\mathrm{T}
$$

$$
= \begin{pmatrix}
\frac{\exp(\frac{\boldsymbol{q}^{(i)}\cdot\boldsymbol{k}^{(1)}}{\sqrt{d_\mathcal{K}}})}{\sum_{\ell=1}^{N}\exp(\frac{\boldsymbol{q}^{(i)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_\mathcal{K}}})} & \cdots & \frac{\exp(\frac{\boldsymbol{q}^{(i)}\cdot\boldsymbol{k}^{(j)}}{\sqrt{d_\mathcal{K}}})}{\sum_{\ell=1}^{N}\exp(\frac{\boldsymbol{q}^{(i)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_\mathcal{K}}})} & \cdots & \frac{\exp(\frac{\boldsymbol{q}^{(i)}\cdot\boldsymbol{k}^{(N)}}{\sqrt{d_\mathcal{K}}})}{\sum_{\ell=1}^{N}\exp(\frac{\boldsymbol{q}^{(i)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_\mathcal{K}}})}
\end{pmatrix}
\begin{pmatrix}
(\boldsymbol{v}^{(1)})^\mathrm{T} \\
\vdots \\
(\boldsymbol{v}^{(j)})^\mathrm{T} \\
\vdots \\
(\boldsymbol{v}^{(N)})^\mathrm{T}
\end{pmatrix}
$$

$$
= \sum_{j=1}^{N}\frac{\exp(\frac{\boldsymbol{q}^{(i)}\cdot\boldsymbol{k}^{(j)}}{\sqrt{d_\mathcal{K}}})}{\sum_{\ell=1}^{N}\exp(\frac{\boldsymbol{q}^{(1)}\cdot\boldsymbol{k}^{(\ell)}}{\sqrt{d_\mathcal{K}}})}(\boldsymbol{v}^{(j)})^\mathrm{T} \in \mathbb{R}^{d_\mathcal{V}}
$$

Advantages and disadvantages of memory-efficient attention show as follows:

- Advantages:
  - $\mathrm{O}(N)$ space complexity, not $\mathrm{O}(N^2)$.
- Disadvantages:
  - Less efficient than parallel computing for multiple $\boldsymbol{q}$.

Where do $\boldsymbol{Q}^{(h)}$, $\boldsymbol{K}^{(h)}$ and $\boldsymbol{V}^{(h)}$ come from? Before multi-head attention layer, we are only given a single tensor of activations $\boldsymbol{A} \in \mathbb{R}^{B\times N\times d_\mathcal{X}}$ - e.g., tokens' embeddings of a sequence. To build $\boldsymbol{Q}^{(h)} \in \mathbb{R}^{B\times N\times d_\mathcal{K}}$, $\boldsymbol{K}^{(h)} \in \mathbb{R}^{B\times N\times d_\mathcal{K}}$

and $\boldsymbol{V}^{(h)} \in \mathbb{R}^{B\times N\times d_{\mathcal{V}}}$ from $\boldsymbol{A} \in \mathbb{R}^{B\times N\times d_{\mathcal{X}}}$, the Transformer uses 3 different linear layers projecting $\boldsymbol{A}$ into $\boldsymbol{Q}^{(h)}$, $\boldsymbol{K}^{(h)}$ and $\boldsymbol{V}^{(h)}$ with different parameters $\boldsymbol{W}_{\mathrm{Q}}^{(h)} \in \mathbb{R}^{d_{\mathcal{X}}\times d_{\mathcal{K}}}$, $\boldsymbol{W}_{\mathrm{K}}^{(h)} \in \mathbb{R}^{d_{\mathcal{X}}\times d_{\mathcal{K}}}$ and $\boldsymbol{W}_{\mathrm{V}}^{(h)} \in \mathbb{R}^{d_{\mathcal{X}}\times d_{\mathcal{V}}}$:

$$\begin{aligned}\boldsymbol{Q}^{(h)} &= \boldsymbol{A}\boldsymbol{W}_{\mathrm{Q}}^{(h)} \in \mathbb{R}^{B\times N\times d_{\mathcal{K}}}\\ \boldsymbol{K}^{(h)} &= \boldsymbol{A}\boldsymbol{W}_{\mathrm{K}}^{(h)} \in \mathbb{R}^{B\times N\times d_{\mathcal{K}}}\\ \boldsymbol{V}^{(h)} &= \boldsymbol{A}\boldsymbol{W}_{\mathrm{V}}^{(h)} \in \mathbb{R}^{B\times N\times d_{\mathcal{V}}}\end{aligned}$$

Now in order to simplify the Transformer model, we use a shared-QK Transformer, which queries and keys are identical:

$$\boldsymbol{q}^{(h,i)} = \boldsymbol{k}^{(h,i)},\ \forall i \in \{1,\cdots,N\}$$

namely we use a shared parameter matrix $\boldsymbol{W}_{\mathrm{QK}}^{(h)} \in \mathbb{R}^{d_{\mathcal{X}}\times d_{\mathcal{K}}}$ for both queries and keys:

$$\begin{aligned}\boldsymbol{Q}^{(h)} &= \boldsymbol{A}\boldsymbol{W}_{\mathrm{QK}}^{(h)} \in \mathbb{R}^{B\times N\times d_{\mathcal{K}}}\\ \boldsymbol{K}^{(h)} &= \boldsymbol{A}\boldsymbol{W}_{\mathrm{QK}}^{(h)} \in \mathbb{R}^{B\times N\times d_{\mathcal{K}}}\end{aligned}$$

It turns out that sharing QK does not affect the performance of Transformer[43].

Hashing attention. We are actually only interested in softmax($\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_{\mathcal{K}}}}$), and softmax is dominated by the largest elements (e.g., $[10^3, 10^3, 10^1, 10^1]$ dominated by $[10^3, 10^3]$),

21.3.4 Cognition

Reading Materials.

- Reformer: The Efficient Transformer (Kitaev et al., 2020)[43].

22 Generative Adversarial Networks (GANs)

22.1 Vanilla GANs

23 Graph Neural Networks (GNNs)

24 Transfer Learning

25 Network Optimization and Regularization

25.1 Network Initialization

25.2 Network Normalization

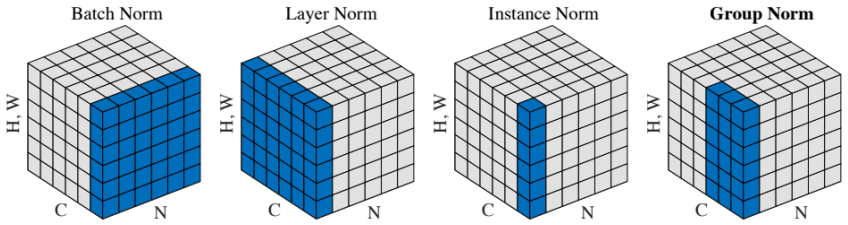


Figure 34: **Normalization methods**. Each subplot shows a feature map tensor, with N as the batch axis, C as the channel axis, and(H, W) as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels. Source: (Wu et al., 2018)[12].

25.2.1 Batch Normalization

Algorithm 17: Batch normalization transformation for 2D-tensor input (tabular explanatory input)[45].

Input: $x \in \mathbb{R}^{N \times d}$: Input of a mini-batch dataset;
ϵ: A small constant;
$\gamma \in \mathbb{R}^d$, $\beta \in \mathbb{R}^d$: learnable affine parameters.
Output: $y = \mathbf{BN}_{\gamma,\beta}(x),\ y \in \mathbb{R}^{N \times d}$.

$\mu_j \leftarrow \frac{1}{N} \sum_{i=1}^{N} x_{ij}$ ▷ Calculate per-dimension mean $\mu_j\ \forall j \in \{1, \dots d\}$

$\sigma_j^2 \leftarrow \frac{1}{N} \sum_{i=1}^{N} (x_{ij} - \mu_j)^2$ ▷ Calculate per-dimension variance: $\sigma_j^2\ \forall j \in \{1, \dots d\}$

$\widehat{x}_{ij} \leftarrow \frac{x_{ij} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$ ▷ Normalize for each element inside mini-batch

$y_{ij} \leftarrow \gamma_j \cdot \widehat{x}_{ij} + \beta_j \equiv \mathbf{BN}_{\gamma,\beta}(x_{ij})\ \forall j \in \{1, \dots d\}$ ▷ Scale & Shift

Algorithm 18: Batch normalization transformation for 4D-tensor input (image explanatory input)[45].

Input: $x \in \mathbb{R}^{N \times C \times H \times W}$: Input of a mini-batch dataset;
ϵ: A small constant;
$\gamma \in \mathbb{R}^C$, $\beta \in \mathbb{R}^C$: learnable affine parameters.
Output: $y = \mathbf{BN}_{\gamma,\beta}(x),\ y \in \mathbb{R}^{N \times C \times H \times W}$.

$\mu_c \leftarrow \frac{1}{N \times H \times W} \sum_{k=1}^{N} \sum_{i=1}^{H} \sum_{j=1}^{W} x_{kcij}$ ▷ Calculate inside per-channel mean $\mu_c\ \forall c \in \{1, \dots C\}$

$\sigma_c^2 \leftarrow \frac{1}{N \times H \times W} \sum_{k=1}^{N} \sum_{i=1}^{H} \sum_{j=1}^{W} (x_{kcij} - \mu_c)^2$ ▷ Calculate inside per-channel variance: $\sigma_c^2\ \forall c \in \{1, \dots C\}$

$\widehat{x}_{nchw} \leftarrow \frac{x_{nchw} - \mu_c}{\sqrt{\sigma_c^2 + \epsilon}}$ ▷ Normalize for each element inside mini-batch

$y_{nchw} \leftarrow \gamma_c \cdot \widehat{x}_{nchw} + \beta_c \equiv \mathbf{BN}_{\gamma,\beta}(x_{nchw})\ \forall c \in \{1, \dots C\}$ ▷ Scale & Shift

Implementation: TensorFlow.

```
# Copyright (C) 2019 Tong Jia. All rights reserved.
"""Definitions of Batch Normalization (BN) operations, both synchronized version and unsynchronized
                                                    version."""
import tensorflow as tf


def batch_norm_2d_unsynchronized_fn(x, training, decay=0.99, epsilon=1e-4, dtype=tf.float32, name="BN"):
    """Definition of unsynchronized barch normalization operation for 2D tensor(e.g. mini-batch images)
                                                    with shape
    [N, C].
    Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift[Ioffe et
                                                    al., 2015]
    (https://arxiv.org/pdf/1502.03167.pdf)
```

```
    Parameters
    ---------------
    :param x: (tensor) -- input tensor with shape [N, D].
                          Note:
                          Dimension(Channel) must be the last dimension of tensor, therefore for mini-batch
                          structured tensor.
    :param training: (bool) -- whether in training mode.
    :param decay: (Optional, float) -- decay rate of exponential moving average for mini-batch mean and variance.
    :param epsilon: (Optional, float) -- small float preventing the denominator of normalization term from being 0.
    :param dtype: (Optional, tf.Dtype) -- value type of tensor inside operation scope.
    :param name: (Optional, str) -- name of specific `BN` variable scope.

    Returns
    ---------------
    :return: (tensor) -- [N, D] tensor with the same shape as input tensor.

    Usage Instance
    ---------------
    tf.set_random_seed(seed=2020)

    x = tf.constant(value=[[2, 4, 6, 8, 10], [1, 3, 5, 7, 9]], dtype=tf.float32)
    x_bn_train = batch_norm_2d_unsynchronized_fn(x=x, trainer=True)  # [2, 5] shape tensor
    x_bn_infer = batch_norm_2d_unsynchronized_fn(x=x, trainer=False)

    with tf.Session() as sess:
        sess.run(fetches=tf.global_variables_initializer())
        r_train = sess.run(fetches=x_bn_train)
        r_infer = sess.run(fetches=x_bn_infer)
    """
    shape = x.get_shape().as_list()
    if len(shape) != 2:
        raise ValueError("invalid input dimension, it must be 2")
    num_units = shape[-1]  # namely the column numbers in a mini-batch data matrix.

    with tf.variable_scope(name_or_scope=name, reuse=False):
        # Define trainable variables
        scale = tf.get_variable(name="scale",
                                shape=[num_units],
                                dtype=dtype,
                                initializer=tf.ones_initializer(dtype=dtype),
                                regularizer=None,
                                trainable=True)  # corresponds to `gamma` in raw paper
        offset = tf.get_variable(name="offset",
                                 shape=[num_units],
                                 dtype=dtype,
                                 initializer=tf.zeros_initializer(dtype=dtype),
                                 regularizer=None,
                                 trainable=True)  # corresponds to `beta` in raw paper
        # Accumulative (EMA) mean and variance
        pop_mean = tf.get_variable(name="pop_mean",
                                   shape=[1, num_units],
                                   dtype=dtype,
                                   initializer=tf.zeros_initializer(dtype=dtype),
                                   regularizer=None,
                                   trainable=False)
        pop_var = tf.get_variable(name="pop_var",
                                  shape=[1, num_units],
                                  dtype=dtype,
                                  initializer=tf.ones_initializer(dtype=dtype),
                                  regularizer=None,
                                  trainable=False)

        def batch_stat():
            """Definition of mini-batch statistics. Used for trainer mode."""
            # Compute mini-batch mean and variance for each column (along the rows)
            batch_mean, batch_var = tf.nn.moments(x=x, axes=[0], keep_dims=True)  # [1, D]
            # Update accumulative mean and variance firstly
            assign_pop_mean_op = tf.assign(ref=pop_mean,
                                           value=pop_mean * decay + batch_mean * (1 - decay),
                                           name="assign_mean_op")
            assign_pop_var_op = tf.assign(ref=pop_var,
                                          value=pop_var * decay + batch_var * (1 - decay),
                                          name="assign_var_op")

            with tf.control_dependencies(control_inputs=[assign_pop_mean_op, assign_pop_var_op]):
                return scale * (x - batch_mean) / tf.sqrt(x=batch_var + epsilon) + offset  # [N, D] shape tensor
```

```
        def pop_stat():
            """Definition of population statistics. Used for evaluation and inference mode."""
            return scale * (x - pop_mean) / tf.sqrt(x=pop_var + epsilon) + offset  # [N, D] shape tensor

    return tf.cond(pred=tf.constant(value=training, dtype=tf.bool), true_fn=batch_stat, false_fn=pop_stat)


def batch_norm_4d_unsynchronized_fn(x, training, decay=0.99, epsilon=1e-4, dtype=tf.float32, name="BN"):
    """Definition of unsynchronized barch normalization operation for 4D tensor(e.g. mini-batch images)
                                                                with shape
    [N, H, W, C].
    Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift[Ioffe et
                                                                al., 2015]
    (https://arxiv.org/pdf/1502.03167.pdf)

    Parameters
    ---------------
    :param x: (tensor) -- input tensor with shape [N, H, W, C].
                          Note:
                          Channel dimension must be the last dimension of tensor, therefore for image
                                                                        tensor, only
                          support `NHWC` data format, not support `NCHW` data format.
    :param training: (bool) -- whether in training mode.
    :param decay: (Optional, float) -- decay rate of exponential moving average for mini-batch mean and
                                                                variance.
    :param epsilon: (Optional, float) -- small float preventing the denominator of normalization term from
                                                                 being 0.
    :param dtype: (Optional, tf.Dtype) -- value type of tensor inside operation scope.
    :param name: (Optional, str) -- name of specific `BN` variable scope.

    Returns
    ---------------
    :return: (tensor) -- [N, H, W, C] tensor with the same shape as input tensor.

    Usage Instance
    ---------------
    tf.set_random_seed(seed=2020)

    x = tf.random.uniform(shape=[32, 224, 224, 3], minval=0.0, maxval=10.0, dtype=tf.float32)
    result = batch_norm_4d_unsynchronized_fn(x=x, trainer=True)  # [32, 224, 224, 3] shape tensor

    with tf.Session() as sess:
        sess.run(fetches=tf.global_variables_initializer())
        r = sess.run(fetches=result)
    """
    shape = x.get_shape().as_list()
    if len(shape) != 4:
        raise ValueError("invalid input dimension, it must be 4")
    num_channels = shape[-1]

    with tf.variable_scope(name_or_scope=name, reuse=False):
        # Define trainable variables
        scale = tf.get_variable(name="scale",
                                shape=[num_channels],
                                dtype=dtype,
                                initializer=tf.ones_initializer(dtype=dtype),
                                regularizer=None,
                                trainable=True)  # corresponds to `gamma` in raw paper
        offset = tf.get_variable(name="offset",
                                 shape=[num_channels],
                                 dtype=dtype,
                                 initializer=tf.zeros_initializer(dtype=dtype),
                                 regularizer=None,
                                 trainable=True)  # corresponds to `beta` in raw paper

        # Accumulative (EMA) mean and variance
        pop_mean = tf.get_variable(name="pop_mean",
                                   shape=[1, 1, 1, num_channels],
                                   dtype=dtype,
                                   initializer=tf.zeros_initializer(dtype=dtype),
                                   regularizer=None,
                                   trainable=False)
        pop_var = tf.get_variable(name="pop_var",
                                  shape=[1, 1, 1, num_channels],
                                  dtype=dtype,
                                  initializer=tf.ones_initializer(dtype=dtype),
                                  regularizer=None,
                                  trainable=False)

        def batch_stat():
```

```
            """Definition of mini-batch statistics. Used for trainer mode."""
            # Compute mini-batch mean and variance for each channel (along the dims without channel)
            batch_mean, batch_var = tf.nn.moments(x=x, axes=[0, 1, 2], keep_dims=True)  # [1, 1, 1, C]
            # Update accumulative mean and variance
            assign_pop_mean_op = tf.assign(ref=pop_mean,
                                           value=pop_mean * decay + batch_mean * (1 - decay),
                                           name="assign_mean_op")
            assign_pop_var_op = tf.assign(ref=pop_var,
                                          value=pop_var * decay + batch_var * (1 - decay),
                                          name="assign_var_op")

            with tf.control_dependencies(control_inputs=[assign_pop_mean_op, assign_pop_var_op]):
                return scale * (x - batch_mean) / tf.sqrt(x=batch_var + epsilon) + offset  # [N, H, W, C]
                                                                                           shape tensor

        def pop_stat():
            """Definition of population statistics. Used for evaluation and inference mode."""
            return scale * (x - pop_mean) / tf.sqrt(x=pop_var + epsilon) + offset  # [N, H, W, C] shape
                                                                                     tensor

    return tf.cond(pred=tf.constant(value=training, dtype=tf.bool), true_fn=batch_stat, false_fn=pop_stat)
```

25.2.2 Layer Normalization

25.2.3 Instance Normalization

25.2.4 Group Normalization

25.2.5 Batch-Instance Normalization

25.3 Network Regularization

25.3.1 Dropout

Part V
Reinforcement Learning Algorithms

26 Policy Gradient Methods

26.1 Vanilla Policy Gradient (VPG)

26.1.1 Notation

26.1.2 Definition

For the problem of discrete control, we need to get an optimal policy function with parameters θ:

$$\pi_\theta(\cdot|s) : \mathcal{S} \to \mathbb{R}^{n_\mathcal{A}}$$

where $n_\mathcal{A}$ represents the number of actions, then we can planning by performing when we are given a state s:

$$a^* \leftarrow \arg\max_{a \in \mathcal{A}} \pi_\theta(a|s)$$

26.1.3 Learning

The objective of any reinforcement learning algorithms is maximizing expected return:

$$\max_\theta \mathcal{J}(\theta) = \mathbb{E}_{s_0,a_0,s_1,\dots}\left[\mathcal{G}_0 = \sum_{t=0}^{\infty}\gamma^t\mathcal{R}_t|\pi_\theta,\mathcal{P},\mu\right] \quad (\mathcal{R}_t = \mathcal{R}(s_t,a_t,s_{t+1}), \mathcal{G}_0 = \mathcal{G}\big((s_0,a_0,s_1,\dots)\big))$$

$$\Rightarrow \max_\theta \mathcal{J}(\theta) = \mathbb{E}_{\tau\sim\rho(\tau;\pi_\theta,\mathcal{P},\mu)}\left[\mathcal{G}(\tau) = \sum_{t=0}^{\infty}\gamma^t\mathcal{R}_t^\tau\right]$$

$$\Rightarrow \max_\theta \mathcal{J}(\theta) = \int_{\tau\in\mathcal{T}}\mathcal{G}(\tau)\rho(\tau;\pi_\theta,\mathcal{P},\mu)d\tau \quad (\mathcal{T} \text{ is the trajectory space})$$

if we want to use gradient-based optimization algorithms to train objective $\mathcal{J}(\theta)$, we need to compute gradients w.r.t. stochastic objective $\mathcal{J}(\theta)$:

$$\begin{aligned}\nabla_\theta\mathcal{J}(\theta) &= \nabla_\theta\int_{\tau\in\mathcal{T}}\mathcal{G}(\tau)\rho(\tau;\pi_\theta,\mathcal{P},\mu)d\tau \\ &= \int_{\tau\in\mathcal{T}}\mathcal{G}(\tau)\nabla_\theta\rho(\tau;\pi_\theta,\mathcal{P},\mu)d\tau \\ &= \int_{\tau\in\mathcal{T}}\mathcal{G}(\tau)\rho(\tau;\pi_\theta,\mathcal{P},\mu)\frac{\nabla_\theta\rho(\tau;\pi_\theta,\mathcal{P},\mu)}{\rho(\tau;\pi_\theta,\mathcal{P},\mu)}d\tau \quad (\rho(\tau;\pi_\theta,\mathcal{P},\mu) > 0,\ \forall\tau\in\mathcal{T})\end{aligned}$$

$$\because\ \rho(\tau;\pi_\theta,\mathcal{P},\mu)\nabla_\theta\log\rho(\tau;\pi_\theta,\mathcal{P},\mu) = \rho(\tau;\pi_\theta,\mathcal{P},\mu)\frac{1}{\rho(\tau;\pi_\theta,\mathcal{P},\mu)}\nabla_\theta\rho(\tau;\pi_\theta,\mathcal{P},\mu) = \nabla_\theta\rho(\tau;\pi_\theta,\mathcal{P},\mu) \quad \textbf{(key point!)}$$

$$\begin{aligned}\therefore\ \nabla_\theta\mathcal{J}(\theta) &= \int_{\tau\in\mathcal{T}}\mathcal{G}(\tau)\rho(\tau;\pi_\theta,\mathcal{P},\mu)\nabla_\theta\log\rho(\tau;\pi_\theta,\mathcal{P},\mu)d\tau \\ &= \mathbb{E}_{\tau\sim\rho(\tau;\pi_\theta,\mathcal{P},\mu)}\left[\mathcal{G}(\tau)\nabla_\theta\log\rho(\tau;\pi_\theta,\mathcal{P},\mu)\right]\end{aligned}$$

According to the properties of MDPs, we can decompose the probability density function of trajectory $\rho(\tau;\pi_\theta,\mathcal{P},\mu)$ along the timesteps of trajectory τ:

$$\begin{aligned}\because \tau &= (s_0^\tau, a_0^\tau, s_1^\tau, a_1^\tau, s_2^\tau, \dots) \\ \therefore \rho(\tau;\pi_\theta,\mathcal{P},\mu) &= \rho\big((s_0^\tau, a_0^\tau, s_1^\tau, a_1^\tau, s_2^\tau, \dots);\pi_\theta,\mathcal{P},\mu\big) \\ &= \mu(s_0^\tau)\,\pi_\theta(a_0^\tau|s_0^\tau)\,\mathcal{P}(s_1^\tau|s_0^\tau,a_0^\tau)\,\pi_\theta(a_1^\tau|s_1^\tau)\,\mathcal{P}(s_2^\tau|s_1^\tau,a_1^\tau)\cdots \quad \text{(properties of MDPs)} \\ &= \mu(s_0^\tau)\prod_{t=0}^{\infty}\pi_\theta(a_t^\tau|s_t^\tau)\prod_{t=0}^{\infty}\mathcal{P}(s_{t+1}^\tau|s_t^\tau,a_t^\tau)\end{aligned}$$

Bring this formula into the above gradients $\nabla_\theta\mathcal{J}(\theta)$ to replace $\rho(\tau;\pi_\theta,\mathcal{P},\mu)$ in $\log$ operation:

$$\begin{aligned}\nabla_\theta\mathcal{J}(\theta) &= \int_{\tau\in\mathcal{T}}\mathcal{G}(\tau)\rho(\tau;\pi_\theta,\mathcal{P},\mu)\nabla_\theta\log\rho(\tau;\pi_\theta,\mathcal{P},\mu)d\tau \\ &= \int_{\tau\in\mathcal{T}}\mathcal{G}(\tau)\rho(\tau;\pi_\theta,\mathcal{P},\mu)\nabla_\theta\log\left(\mu(s_0^\tau)\prod_{t=0}^{\infty}\pi_\theta(a_t^\tau|s_t^\tau)\prod_{t=0}^{\infty}\mathcal{P}(s_{t+1}^\tau|s_t^\tau,a_t^\tau)\right)d\tau\end{aligned}$$

$$
\begin{aligned}
&= \int_{\tau \in \mathcal{T}} \mathcal{G}(\tau) \rho(\tau; \pi_\theta, \mathcal{P}, \mu) \nabla_\theta \left(\log \mu(s_0^\tau) + \log \left(\prod_{t=0}^{\infty} \pi_\theta(a_t^\tau | s_t^\tau) \right) + \log \left(\prod_{t=0}^{\infty} \mathcal{P}(s_{t+1}^\tau | s_t^\tau, a_t^\tau) \right) \right) d\tau \\
&= \int_{\tau \in \mathcal{T}} \mathcal{G}(\tau) \rho(\tau; \pi_\theta, \mathcal{P}, \mu) \nabla_\theta \left(\log \mu(s_0^\tau) + \sum_{t=0}^{\infty} \log \pi_\theta(a_t^\tau | s_t^\tau) + \sum_{t=0}^{\infty} \log \mathcal{P}(s_{t+1}^\tau | s_t^\tau, a_t^\tau) \right) d\tau \\
&= \int_{\tau \in \mathcal{T}} \mathcal{G}(\tau) \rho(\tau; \pi_\theta, \mathcal{P}, \mu) \left(\nabla_\theta \log \mu(s_0^\tau) + \nabla_\theta \left(\sum_{t=0}^{\infty} \log \pi_\theta(a_t^\tau | s_t^\tau) \right) + \nabla_\theta \left(\sum_{t=0}^{\infty} \log \mathcal{P}(s_{t+1}^\tau | s_t^\tau, a_t^\tau) \right) \right) d\tau \\
&= \int_{\tau \in \mathcal{T}} \mathcal{G}(\tau) \rho(\tau; \pi_\theta, \mathcal{P}, \mu) \left(\nabla_\theta \log \mu(s_0^\tau) + \sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t^\tau | s_t^\tau) + \sum_{t=0}^{\infty} \nabla_\theta \log \mathcal{P}(s_{t+1}^\tau | s_t^\tau, a_t^\tau) \right) d\tau
\end{aligned}
$$

Because the agent θ can only adjust its own policy π_θ during the interaction with a given MDP environment $\mathcal{M} =< \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, \mu, \gamma >$, but cannot modify or optimize the environment, therefore the gradient of initial state probability $\mu(s)$ and model dynamics $\mathcal{P}(s'|s,a)$ are both 0, and we can continue to derivate $\nabla_\theta \mathcal{J}(\theta)$ function as follows:

$$
\begin{aligned}
\nabla_\theta \mathcal{J}(\theta) &= \int_{\tau \in \mathcal{T}} \mathcal{G}(\tau) \rho(\tau; \pi_\theta, \mathcal{P}, \mu) \left(\sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t^\tau | s_t^\tau) \right) d\tau \\
&= \mathbb{E}_{\tau \sim \rho(\tau; \pi_\theta, \mathcal{P}, \mu)} \left[\mathcal{G}(\tau) \left(\sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t^\tau | s_t^\tau) \right) \right] \quad \text{(Policy Gradient!)}
\end{aligned}
$$

Equivalently, we can get the primitive objective function $\mathcal{J}(\theta)$ (objective form is not important relative to the gradient form):

$$
\begin{aligned}
&\max_\theta \ \mathcal{J}(\theta) = \mathbb{E}_{\tau \sim \rho(\tau; \pi_\theta, \mathcal{P}, \mu)} \left[\mathcal{G}(\tau) = \sum_{t=0}^{\infty} \gamma^t \mathcal{R}_t^\tau \right] \\
\Leftrightarrow &\max_\theta \ \mathcal{J}(\theta) = \mathbb{E}_{\tau \sim \rho(\tau; \pi_\theta, \mathcal{P}, \mu)} \left[\mathcal{G}(\tau) \left(\sum_{t=0}^{\infty} \log \pi_\theta(a_t^\tau | s_t^\tau) \right) \right]
\end{aligned}
$$

Therefore when we know the specific content of the gradient, we can use any kinds of gradient-based optimizers to learn agent parameters θ by gradient ascent methods (not gradient descent, because maximization), e.g., SGD:

$$\theta \leftarrow \theta + \alpha \nabla_\theta \mathcal{J}(\theta)$$

Algorithm 19: REINFORCE: Monte-Carlo policy gradient control (episodic) for π^*[46].

Input: $\pi(a|s;\theta) : \mathcal{S} \to \mathbb{P}^{|\mathcal{A}|}$: A differentiable policy parameterization;
$\alpha > 0$: Step size;
$\theta \in \mathbb{R}^d$: Initial policy parameters.
Output: θ: Resulting parameters of policy agent.

repeat
  $\tau_\theta = (s_0, a_0, s_1, \mathcal{R}_0, \ldots, s_{T-1}, a_{T-1}, s_T, \mathcal{R}_T)$ ▷ Generate an episode following policy π_θ
  for $t \in \{0, \ldots, T-1\}$ **do**
    $\mathcal{G}_t \leftarrow \sum_{k=t}^{T} \gamma^{k-t} \mathcal{R}_k$ ▷ Return from step t
    $\theta \leftarrow \theta + \alpha \gamma^t \mathcal{G}_t \nabla_\theta \log \pi_\theta(a_t|s_t)$ ▷ Update policy parameters
  end
until *forever*;

Solving $\nabla_\theta \log \pi_\theta(a_t^\tau | s_t^\tau)$ as Classification Task. Here we know the analytical formula for policy gradient as follow:

$$
\begin{aligned}
\nabla_\theta \mathcal{J}(\theta) &= \int_{\tau \in \mathcal{T}} \mathcal{G}(\tau) \rho(\tau; \pi_\theta, \mathcal{P}, \mu) \left(\sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t^\tau | s_t^\tau) \right) d\tau \\
&= \mathbb{E}_{\tau \sim \rho(\tau; \pi_\theta, \mathcal{P}, \mu)} \left[\mathcal{G}(\tau) \left(\sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t^\tau | s_t^\tau) \right) \right]
\end{aligned}
$$

$$\approx \frac{1}{N}\sum_{i=1}^{N}\mathcal{G}(\tau^{(i)})\left(\sum_{t=0}^{T(\tau)}\nabla_\theta \log \pi_\theta(a_t^{(i)}|s_t^{(i)})\right) \quad \text{(i.i.d. sample } N \text{ tarjectories from } \tau\sim\rho(\tau;\pi_\theta,\mathcal{P},\mu))$$

If we are given for sure:

- $s\in\mathbb{R}^{|\mathcal{S}|}$: state representation, no matter continuous or discrete state space
- $a\in\{1,\dots,|\mathcal{A}|\}$: action representation, discrete action space

then we can think of this policy learning problem as a typical classification problem, namely use cross-entropy (or KL-divergence) as objective:

$$\begin{aligned}
&\max_\theta \ \log \pi_\theta(a_t^\tau|s_t^\tau) \quad \text{(log-likelihood)}\\
\Rightarrow &\min_\theta \ \mathrm{KL}\left(p_t^\tau\|\pi_\theta(s_t^\tau)\right) \quad (a_t^\tau\sim p_t^\tau, p_t^\tau\in\mathbb{P}^{|\mathcal{A}|} \text{ is an action selection probability distribution})\\
\Rightarrow &\min_\theta \ -p_t^\tau\cdot\log\pi_\theta(s_t^\tau)\\
\Rightarrow &\min_\theta \ -\sum_{k\in\mathcal{A}} p_{t,k}^\tau \log\left[\pi_\theta(s_t^\tau)\right]_k\\
\Rightarrow &\max_\theta \ \sum_{k\in\mathcal{A}} p_{t,k}^\tau \log\left[\pi_\theta(s_t^\tau)\right]_k
\end{aligned}$$

26.1.4 Variants of Return $\mathcal{G}(\tau;t)$ in Policy Gradient Methods (Not Accomplished)

Based on the previous derivation of policy-based algorithm in 26.1.3, we know the policy gradient is as follow:

$$\begin{aligned}
\nabla_\theta\mathcal{J}(\theta) &= \mathbb{E}_{\tau\sim\rho(\tau;\pi_\theta,\mathcal{P},\mu)}\left[\mathcal{G}(\tau)\left(\sum_{t=0}^{\infty}\nabla_\theta\log\pi_\theta(a_t^\tau|s_t^\tau)\right)\right]\\
&= \mathbb{E}_{\tau\sim\rho(\tau;\pi_\theta,\mathcal{P},\mu)}\left[\left(\sum_{t=0}^{\infty}\mathcal{G}(\tau;t)\nabla_\theta\log\pi_\theta(a_t^\tau|s_t^\tau)\right)\right]\\
&\approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T(\tau^{(i)})-1}\widehat{\mathcal{G}}(\tau^{(i)};t)\nabla_\theta\log\pi_\theta(a_t^\tau|s_t^\tau)
\end{aligned}$$

here we have not specified any specific estimation of return (reward-to-go) $\widehat{\mathcal{G}}(\tau;t)$ (it serves as the independent part of practical stepsize (another part is public part: α) for a specific sample (s_t^τ,a_t^τ) in policy optimization process), so we will specify kinds of specific reward-to-go estimations $\widehat{\mathcal{G}}(\tau;t)$.

We make summary of kinds of return estimation methods in policy gradient method as follows.

Vanilla Monte-Carlo Policy Gradient. return estimation as follows:

$$\begin{aligned}
\widehat{\mathcal{G}}(\tau;t) &= \sum_{t=0}^{T(\tau)-1}\gamma^t\widehat{\mathcal{R}}_t^\tau\\
\widehat{\mathcal{R}}_t^\tau &= \mathcal{R}(s_t^\tau,a_t^\tau,s_{t+1}^\tau) \quad \text{(for a given } \mathcal{M}=<\mathcal{S},\mathcal{A},\mathcal{R},\mathcal{P},\mu,\gamma>,\ \mathcal{R} \text{ is known)}
\end{aligned}$$

corresponding policy gradient as follows:

$$\nabla_\theta\mathcal{J}(\theta) = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T(\tau)-1}\underbrace{\left(\sum_{t=0}^{T(\tau)-1}\gamma^t\widehat{\mathcal{R}}_t^\tau\right)}_{\widehat{\mathcal{G}}(\tau;t)}\nabla_\theta\log\pi_\theta(a_t^\tau|s_t^\tau)$$

Vanilla Monte-Carlo Policy Gradient with Baseline. return estimation as follows:

$$\widehat{\mathcal{G}}(\tau;t) = \sum_{t=0}^{T(\tau)-1}\gamma^t\widehat{\mathcal{R}}_t^\tau$$

baseline estimation as follows:

$$\begin{aligned}\widehat{b}(\tau;t) &= \mathbb{E}_{\tau\sim\rho(\tau;\pi_\theta,\mathcal{P},\mu)}\left[\mathcal{G}(\tau;t)\right] \\ &= \frac{1}{N}\sum_{i=1}^{N}\end{aligned}$$

Table 1: Varients of $\mathcal{G}(\tau;t)$ in policy gradient method.

Varient	**Return Estimation**	**Gradient**
Vanilla Monte-Carlo	$\mathcal{G}(\tau)=\sum_{t=0}^{\infty}\gamma^t\mathcal{R}_t$	$\nabla_\theta\mathcal{J}(\theta)=\mathbb{E}_{\tau\sim\rho(\tau;\pi_\theta,\mathcal{P},\mu)}\left[\left(\sum_{t=0}^{\infty}\gamma^t\mathcal{R}_t\right)\left(\sum_{t=0}^{\infty}\log\pi_\theta(a_t^\tau\|s_t^\tau)\right)\right]$
Vanilla Monte-Carlo Baseline	$\mathcal{G}(\tau)=\sum_{t=0}^{\infty}\gamma^t\mathcal{R}_t-b(\tau)$ $b(\tau)=\mathbb{E}_{\tau\sim\rho(\tau;\pi_\theta,\mathcal{P},\mu)}\left[\gamma^t\mathcal{R}_t\right]$	

26.1.5 Cognition

Opinions.

1. Stepsize of policy gradient method in practical optimization process comes from two parts:
   - $\widehat{\mathcal{G}}(\tau;t)$: return encouragement in trajectory τ after timestep t (after the action selection a_t^τ under state observation s_t^τ). Sample-specific part.
   - α: stepsize of optimizer (e.g., SGD, Adam). Sample-common part.

   we have not only listed some kinds of $\widehat{\mathcal{G}}(\tau;t)$ in previous Subsubsection 26.1.4, but also listed some kinds of α selections in **??**. This two parts of stepsize are not independent in the actual optimization process, but affect each other, especially using adaptive optimizers (i.e., selection of adaptive optimizers will influences the original effect of $\widehat{\mathcal{G}}(\tau;t)$). Therefore we need to propose some innovative optimization methods in order to **keep original effect of** $\widehat{\mathcal{G}}(\tau;t)$ (e.g., put $\widehat{\mathcal{G}}(\tau;t)$ after computing gradient and before applying gradient) and **using adaptive methods** α **to revising (adjusting) effect by** $\widehat{\mathcal{G}}(\tau;t)$. (Inspired by paper Decoupled Weight Decay Regularization[34]).

Reading Materials.

- Slides UC Berkeley: CS285-Policy Gradients.
- Slides Toronto: Learning Reinforcement Learning by Learning REINFORCE.
- Lil'Log: Policy Gradient Algorithms.

27 Value Approximation Methods

27.1 Q-Learning

Part VI
Algorithm Applications

28 Recommender System

29 Click-Through Rate Prediction

29.1 Factorization Machines (FM)

29.1.1 Motivation

Problem: Why need click-through rate prediction algorithms in industry?

- Consumer: The reality is that information is overloaded in platform, and it is very difficult to find the information (e.g., advertisement) you are interested in from a large amount of information without any help from platform.
- Producer: Producers want to make their advertisements stand out in platform, but the reality is that it's hard to get consumers' attention without any help from platform.
- Platform: Platform's profit method is pay-per click from producer.

$$\begin{aligned}\text{Profit} &= \text{Ad clicks} \times \text{Cost per-click} \\ &= \underbrace{\text{Ad displays}}_{\text{const}} \times \underbrace{\text{Click-through rate}}_{\textbf{key!}} \times \underbrace{\text{Cost per-click}}_{\text{const}}\end{aligned}$$

The existing methods and corresponding problems are shown as follows:

- Logistic regression model $\widehat{y}(\boldsymbol{x}) = \frac{1}{1+\exp(-\sum_{j=1}^{D} w_j x_j)}$ assumes all features are independent, therefore it doesn't account for ***feature conjunctions***, which is a very important piror background knowledge for CTR problems.
- Polynomial logistic regression model with inner function $\widehat{z}(\boldsymbol{x}) = \sum_{i=1}^{D} w_i x_i + \sum_{i=1}^{D} \sum_{j=i+1}^{D} w_{i,j} x_i x_j$ aims to modeling feature conjunctions. But it's hard to be optimized by gradient-based mathods when using sparse dataset (e.g., long one-hot encoding vector of each field).

29.1.2 Contribution

29.1.3 Definition

29.1.4 Learning

29.1.5 Cognition

Reading Materials.

- Factorization Machines (Rendle, 2010)[47].
- Factorization Machines with libFM (Rendle, 2012)[48].

29.2 Field-aware Factorization Machines for CTR Prediction (FFM)

29.2.1 Motivation

29.2.2 Contribution

29.2.3 Definition

29.2.4 Learning

29.2.5 Cognition

Reading Materials.

- Field-aware Factorization Machines for CTR Prediction (Juan et al., 2016)[49].
- [Slides] Field-aware Factorization Machines.
- Field-aware Factorization Machines in a Real-world Online Advertising System (Juan et al., 2017)[50].
- One-class Field-aware Factorization Machines for Recommender Systems with Implicit Feedbacks (Yuan et al., 2019)[51].

29.3 Automatic Feature Interaction Learning via Self-Attentive Neural Networks (AutoInt)

29.3.1 Motivation

29.3.2 Contribution

29.3.3 Definition

29.3.4 Cognition

Reading Materials.

- Automatic Feature Interaction Learning via Self-Attentive Neural Networks (Song et al., 2018)[52].

29.4 One-class Field-aware Factorization Machines for Recommender Systems with Implicit Feedbacks (OCFFM)

29.4.1 Notation

Table 2: Notations used for the mathematical derivation of OCFFM in Subsection 29.4.

Notation	Description
D	Number of features.
D_f	Number of features in f-th field.
F	Number of fields.
k	Dimensionality of latent vector of all feature. A pre-specified hyper-parameter of model.
j, j_1, j_2	Feature index.
f_j	Field index which the j-th feature belongs.
$\boldsymbol{x} \in \mathbb{R}^D$	Feature vector containing D feature values.
$\boldsymbol{x}_f \in \mathbb{R}^{D_f}$	A D_f dimensional feature vector belongs to the f-th field.
$x_j \in \mathbb{R}$	A j-th feature value.
$y \in \mathbb{R}$	Ground-truth label.
$\boldsymbol{v}_j \in \mathbb{R}^k$	A k-dimensional Latent vector of the j-th feature. Notation used for FM 29.1.
$\boldsymbol{v}_{j_1,f_{j_2}} \in \mathbb{R}^k$	A k-dimensional latent vector of the feature j_1, which learns the latent effect with the interacted features that belong to the field f_{j_2}. Notation used for FFM 29.2.
$V^{(f_1,f_2)} \in \mathbb{R}^{D_{f_1} \times k}$	Embedding matrix of field f_1 for encoding conjunction with field f_2, i.e., all feature embeddings belong to field f_1 and to be conjuncted (i.e., interacted) with field f_2.

29.4.2 Motivation

- **One-class Matrix Factorization (OCMF).** Recommender systems with implicit feedbacks is a typical one-class scenario, where only positive labels are available. Such ***positive-unlabeled (PU) learning*** problems can be solved by one-class matrix factorization (OCMF) (i.e. user embedding and item embeding).
- **One-class Matrix Factorization with Side Information (OCMFSI).** Recently, OCMF with side information (OCMFSI) on users and items has been proposed as a powerful extension of OCMF.
- **Factorization Machines (FM).** OCMFSI is strongly related to Factorization Machines (FM), which is a general classification and regression model.
- **Field-aware Factorization Machines (FFM).** FFM extends FM by considering the field information, but it's slow and inefficient to be trained on large-scale datasets.

29.4.3 Contribution

- Propose a novel model (i.e., OCFFM) for ***positive-unlabeled (PU) learning*** problems.
- Develop an efficient optimization algorithm such that OCFFM can be trained on the large-scale datasets.

29.4.4 Definition

Review of FM and FFM. The inference formulation of FM (Rendle, 2010)[47] and FFM (Juan et al., 2016)[49] are shown as follows:

- **Factorization Machines (FM).**

$$\widehat{y}_{\text{FM}}(\boldsymbol{x}) = \sum_{j_1=1}^{D} \sum_{j_2=j_1+1}^{D} (\boldsymbol{v}_{j_1}^{\text{T}} \boldsymbol{v}_{j_2}) x_{j_1} x_{j_2}$$

- **Field-aware Factorization Machines (FFM).**

$$\widehat{y}_{\text{FFM}}(\boldsymbol{x}) = \sum_{j_1=1}^{D} \sum_{j_2=j_1+1}^{D} (\boldsymbol{v}_{j_1,f_{j_2}}^{\text{T}} \boldsymbol{v}_{j_2,f_{j_1}}) x_{j_1} x_{j_2}$$

where $\boldsymbol{v}_{j_1}^{\text{T}} \boldsymbol{v}_{j_2} \in \mathbb{R}$ and $\boldsymbol{v}_{j_1,f_{j_2}}^{\text{T}} \boldsymbol{v}_{j_2,f_{j_1}} \in \mathbb{R}$ means the ***coefficient of feature conjunctions*** (i.e., score of feature interaction).

Rewrite FFM. Specifically, the feature vector $\boldsymbol{x} \in \mathbb{R}^D$ is considered as a concatenation of F sub-vectors, i.e., each field f corresponds to a sub-vector

$$\boldsymbol{x} = \begin{bmatrix} \boldsymbol{x}_1 \\ \vdots \\ \boldsymbol{x}_f \\ \vdots \\ \boldsymbol{x}_F \end{bmatrix} \in \mathbb{R}^D$$

where $\boldsymbol{x}_f \in \mathbb{R}^{D_f}$ includes D_f features which belong to the f-th field. Therefore, above FFM formulation can be written as follows:

$$\widehat{y}(\boldsymbol{x}) = \sum_{f_1=1}^{F} \sum_{f_2=1}^{F} \left((V^{(f_1,f_2)})^{\mathrm{T}} \boldsymbol{x}_{f_1} \right)^{\mathrm{T}} \left((V^{(f_2,f_1)})^{\mathrm{T}} \boldsymbol{x}_{f_2} \right)$$

where

$$V^{(f_1,f_2)} = [\boldsymbol{w}_{1,f_2}, \cdots, \boldsymbol{w}_{D_{f_1},f_2}]^{\mathrm{T}} \in \mathbb{R}^{D_{f_1} \times k}$$

represents the embedding matrix of the features in field f_1, encoding their interactions to the features in field f_2. And for a field j_1, $(V^{(f_1,f_2)})^{\mathrm{T}} \boldsymbol{x}_{f_1} \in \mathbb{R}^k$

29.4.5 Cognition

Reading Materials.

- One-class Field-aware Factorization Machines for Recommender Systems with Implicit Feedbacks (Yuan et al., 2019)[51].

30 Object Detection

30.1 You Only Look Once: Unified, Real-Time Object Detection (YOLO)

30.1.1 Notation

30.1.2 Motivation

- Existed object detection models perform inference too slow.

30.1.3 Contribution

30.1.4 Definition

Methodology.

Network Architecture.

Objective.

30.1.5 Continuation

30.1.6 Cognition

Reading Materials.

- You Only Look Once: Unified, Real-Time Object Detection (Redmon et al., 2015)[53].
- (Slides) You Only Look Once: Real-time Detecrion CVPR-2016.
- YOLO9000: Better, Faster, Stronger (Redmon and Farhadi, 2016)[54].
- YOLOv3: An Incremental Improvement (Redmon and Farhadi, 2018)[55].

30.2 Single Shot MultiBox Detector (SSD)

30.2.1 Motivation

30.2.2 Contribution

30.2.3 Definition

30.2.4 Cognition

Reading Materials.

- SSD: Single Shot MultiBox Detector (Liu et al., 2015)[56].

31 Image Segmentation

32 Image Super-Resolution

33 Image Translation

33.1 Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks (CycleGAN)

33.1.1 Motivation

- For many image-to-image translation tasks, paired training data will not be available. We therefore need to seek an algorithm that can learn to translate between domains without paired input-output examples.
- Many image-to-image translation tasks are under the assumption that, there is some underlying relationship between the domains - for example, that they are two different renderings of the same underlying scene - and seek to learn that relationship.

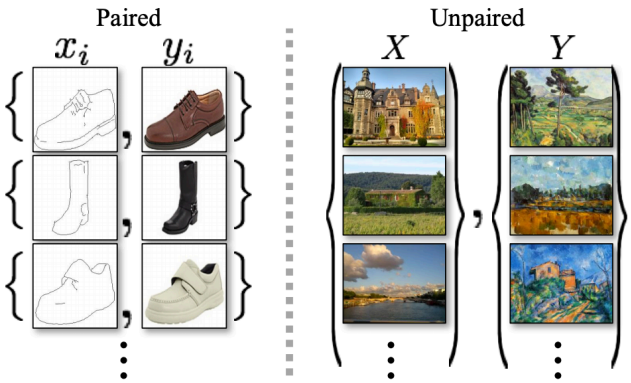


Figure 35: Paired training data (left) consists of training examples $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{N}$, where the correspondence between $x^{(i)}$ and $y^{(i)}$ exists. We instead consider unpaired training data (right), consisting of a source set $\mathcal{D}_{\mathcal{X}} = \{x^{(i)}\}_{i=1}^{N_{\mathcal{X}}}$ and a target set $\mathcal{D}_{\mathcal{Y}} = \{y^{(j)}\}_{j=1}^{N_{\mathcal{Y}}}$ with no information provided as to which $x^{(i)}$ matches which $y^{(j)}$. Source: (Zhu et al., 2017)[13].

33.1.2 Contribution

- Present a method that can learn to capture special characteristics of one image collection and figuring out how these characteristics could be translated into the other image collection, all ***in the absence of any paired training examples***.

33.1.3 Definition

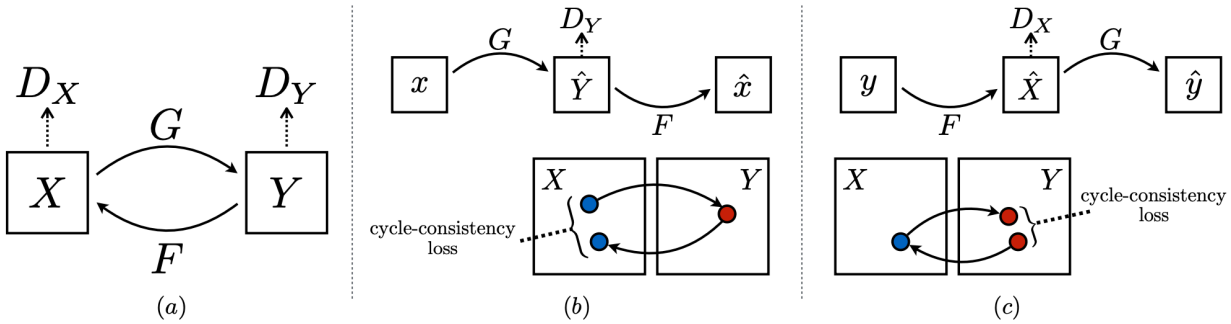


Figure 36: (a) CyclaGAN (Zhu et al., 2017)[13] model contains two mapping functions (i.e., two generators) $G : X \mapsto Y$ and $F : Y \mapsto X$, and associated adversarial discriminators D_Y and D_X. D_Y encourages G to translate X into outputs indistinguishable from domain Y, and vice versa for D_X and F. To further regularize the mappings, we introduce two ***cycle consistency losses*** that capture the intuition that if we translate from one domain to the other and back again we should arrive at where we started: (b) forward cycle-consistency loss $x \mapsto G(x) \mapsto F(G(x)) \approx x$, and (c) backward cycle-consistency loss $y \mapsto F(y) \mapsto G(f(y)) \approx y$. Source: (Zhu et al., 2017)[13].

Adversarial Loss.

Cycle Consistency Loss.

Full Objective.

33.1.4 Cognition

Reading Materials.

- Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks (Zhu et al., 2017)[13].

33.2 Unified Generative Adversarial Networks for Multi-Domain Image-to-Image Translation (StarGAN)

33.2.1 Motivation

33.2.2 Contribution

33.2.3 Definition

33.2.4 Cognition

Reading Materials.

- StarGAN: Unified Generative Adversarial Networks for Multi-Domain Image-to-Image Translation (Choi et al., 2018)[57].

34 Pre-Trained Language Models

34.1 Distributed Representations of Words and Phrases and their Compositionality (Word2Vec)

34.1.1 Notation

All mathematical notations used for the derivation of language models are shown in following Table 3.

Table 3: Notations used for the mathematical derivation of language models in Section 34.

Notation	Description
$n_{\mathcal{V}}$	Vocabulary size, i.e., the number of distinctive tokens in corpus.
$n_{\mathcal{B}}$	Number of sequences in mini-batch subset $\mathcal{D}_{\mathcal{B}}$ of corpus $\mathcal{D}$.
$n_{\mathcal{D}}$	Number of sequences in corpus $\mathcal{D}$.
$n_{\mathcal{U}}$	Length of the sequence $\mathcal{U}$.
$n_{\mathcal{C}}$	Number of classes used in supervised fine-tuning phase.
$d_{\mathcal{X}}$	Dimensionality of token embedding.
$u \in \{1, \cdots, n_{\mathcal{V}}\}$	A token index.
$\boldsymbol{x} \in \mathbb{R}^{d_{\mathcal{X}}}$	Embedding vector of token u.
$u_j^{(i)} \mapsto \boldsymbol{x}_j^{\mathcal{U}_i} \in \mathbb{R}^{d_{\mathcal{X}}}$	Embedding lookup for j-th token in sequence $\mathcal{U}_i$.
$\mathcal{V}$	Vocabulary of corpus, i.e., the set of distinctive tokens.
$\mathcal{D} = \{\mathcal{U}_i\}_{i=1}^{n_{\mathcal{D}}}$	Training corpus, containing $n_{\mathcal{D}}$ training sequences.
$\mathcal{D}_{\mathcal{B}} = \{\mathcal{U}_i\}_{i=1}^{n_{\mathcal{B}}} \sim \mathcal{D}$	Mini-batch sequences sampled from corpus $\mathcal{D}$.
$\mathcal{U}_i = (u_1^{(i)}, \cdots, u_j^{(i)}, \cdots, u_{n_{\mathcal{U}}}^{(i)})$	The i-th sequence with multi tokens in corpus $\mathcal{D}$.
$\mathcal{M}_i$	Set of masked token indices in sequence $\mathcal{U}_i$.
$\mathcal{U}_{\mathcal{M}_i}$	Produced masked sequence of original sequence $\mathcal{U}_i$ masked by the set $\mathcal{M}_i$.
$\mathcal{Z}_{n_{\mathcal{U}}}$	The set of all possible permutations of the $n_{\mathcal{U}}$-length index sequence $(1, 2, \cdots, n_{\mathcal{U}})$, used in Subsection 34.9.
$\mathcal{I} \sim \mathcal{Z}_{n_{\mathcal{U}}}$	A $n_{\mathcal{U}}$-length index sequence sampled from the set $\mathcal{Z}_{n_{\mathcal{U}}}$, $\mathcal{I}_j$ is the j-th element (i.e., a index) in sequence $\mathcal{I}$, used in Subsection 34.9.
$u_{j':1 \leq j' \leq j-1} = (u_1, \cdots, u_{j-1})$	A subsequence of $\mathcal{U}$ which contains tokens before j-th position.
L	Number of layers (e.g, Transformer blocks or LSTMs blocks).
$\ell \in \{1, \cdots, L\}$	Index for indicating each basic block.
H_ℓ	Number of self-attention heads in l-th Transformer block.
$h \in \{1, \cdots, H_\ell\}$	Head index.
$i \in \{1, \cdots, n_{\mathcal{D}}\}$	Index for indicating each sequence $\mathcal{U}_i$ in $\mathcal{D}$.
$j \in \{1, \cdots, n_{\mathcal{U}}\}$	Index for indicating each token u_j in $\mathcal{U}$.
$k \geq 1$	Sliding contextual window size.
$\Theta_{\mathcal{X}} \in \mathbb{R}^{n_{\mathcal{V}} \times d_{\mathcal{X}}}$	Embedding matrix for all distinctive tokens in corpus $\mathcal{D}$.
$\Theta_{\mathcal{P}} \in \mathbb{R}^{n_{\mathcal{U}_{\max}} \times d_{\mathcal{X}}}$	Positional encoding, $n_{\mathcal{U}_{\max}}$ represents the maximum length of sequence in corpus.
$\Theta_{\mathcal{S}} \in \mathbb{R}^{d_L \times n_{\mathcal{V}}}$	Final linear projection parameters used for added softmax layer, d_L represents the dimensionality of L-th encoding layer (i.e., the last encoding layer).
$\boldsymbol{X}_{\mathcal{U}_i} \in \mathbb{R}^{n_{\mathcal{U}_i} \times d_{\mathcal{X}}}$	Token embeddings of all tokens in sequence $\mathcal{U}_i$ after lookup opeartion. It is used in Transformer-based models.
$\boldsymbol{H}_{\mathcal{U}_i}^{(\ell)} \in \mathbb{R}^{n_{\mathcal{U}_i} \times d_\ell}$	Encoding representations for all tokens inside sequence $\mathcal{U}_i$ in ℓ-th hidden layer. It is used in Transformer-based models.
$\boldsymbol{x}_c^{\mathcal{V}} \in \mathbb{R}^{d_{\mathcal{X}}}$	the c-th line in $\Theta_{\mathcal{X}}$, namely the embedding for c-th distinctive token in vocabulary $\mathcal{V}$.
$\boldsymbol{x}_j^{\mathcal{U}_i} \in \mathbb{R}^{d_{\mathcal{X}}}$	The j-th row of $\boldsymbol{X}_{\mathcal{U}_i}$, representing the token embedding of j-th token in sequence $\mathcal{U}_i$.
$\boldsymbol{h}_{j,\ell}^{\mathcal{U}} \in \mathbb{R}^{d_\ell}$	The j-th row of $\boldsymbol{H}_{\mathcal{U}}^{(\ell)}$, corresponds to the j-th token in sequence $\mathcal{U}$.
$\boldsymbol{h}_{j,\ell}^{(i)} \in \mathbb{R}^{d_\ell}$	The j-th row of $\boldsymbol{H}_{\mathcal{U}_i}^{(\ell)}$, corresponds to the j-th token in sequence $\mathcal{U}_i$.

34.1.2 Motivation

- One-hot encoding for each token takes huge dimensionality, i.e., vector dimension = number of words in vocabulary (e.g., 500000).
- All one-hot encodings are orthogonal with others and have the same distance with others, i.e., $\sqrt{2}$ as Euclidean distance. Therefore, there is no natural notion of **similarity** for one-hot vectors!

$$\text{model} = [0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0]^{\mathrm{T}} \in \mathbb{R}^{n_{\mathcal{V}}}$$
$$\text{hotel} = [0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0]^{\mathrm{T}} \in \mathbb{R}^{n_{\mathcal{V}}}$$

- Consider to encode each token u with a low-dimensional dense vector $\boldsymbol{x} \in \mathbb{R}^{d_{\mathcal{X}}}$, i.e., ***distributed representation*** where $d_{\mathcal{X}} \ll n_{\mathcal{V}}$ and also containing semantic similarity inside distance of embedding space.

$$\text{model} = \begin{pmatrix} 0.286 \\ 0.793 \\ -1.774 \\ -0.107 \end{pmatrix} \in \mathbb{R}^{d_{\mathcal{X}}} \quad \text{hotel} = \begin{pmatrix} 0.349 \\ -0.271 \\ 0.183 \\ 1.121 \end{pmatrix} \in \mathbb{R}^{d_{\mathcal{X}}}$$

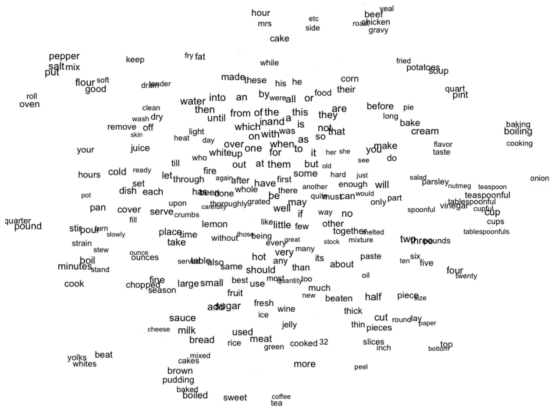


Figure 37: Illustrations for a toy distributed representations $\boldsymbol{x}$ for all distinctive tokens in corpus $u \in \mathcal{D}$. Source: https://www.adityathakker.com/introduction-to-word2vec-how-it-works/.

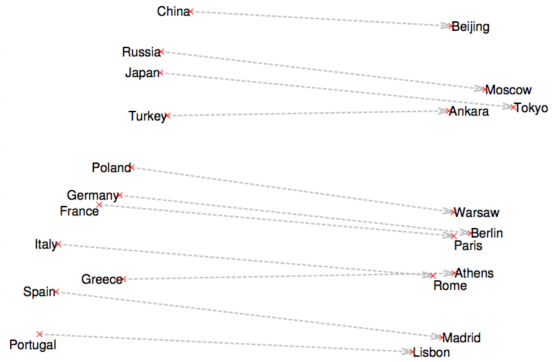


Figure 38: Similar semantic meanings between (u_1, u_2) and (u_3, u_4) have similar embedding distances $\|\boldsymbol{x}_1 - \boldsymbol{x}_2\|$ and $\|\boldsymbol{x}_3 - \boldsymbol{x}_4\|$. Source: https://algorithmia.com/algorithms/nlp/Word2Vec/docs.

34.1.3 Contribution

- Introduce two model variants for learning word distributed representations, where $k > 0$ represents the ***sliding contextual window size***.
  - ***Skip-gram (SG)***: $\Pr(u_{j-k}, \cdots, u_{j-1}, u_{j+1}, \cdots, u_{j+k} | u_j)$ Predict context (i.e., outside) words (position independent) given center word.

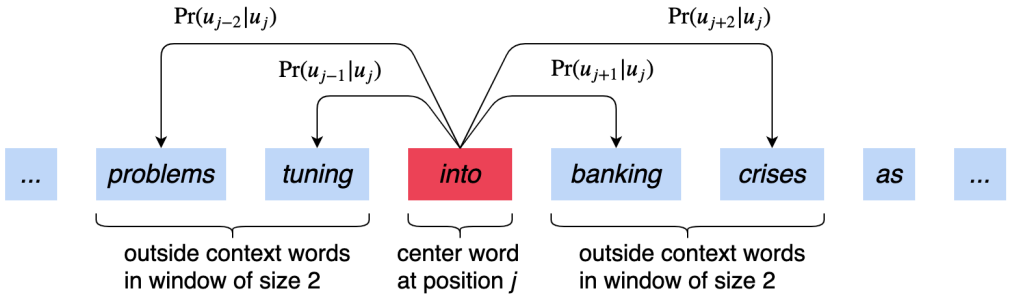


Figure 39: Example windows and process for computing $\Pr(u_{j+k}|u_j)$, i.e., Skip-gram method.

  - ***Continuous Bag of Words (CBOW)***: $\Pr(u_j | u_{j-k}, \cdots, u_{j-1}, u_{j+1}, \cdots, u_{j+k})$ Predict center word from (bag of) context words.

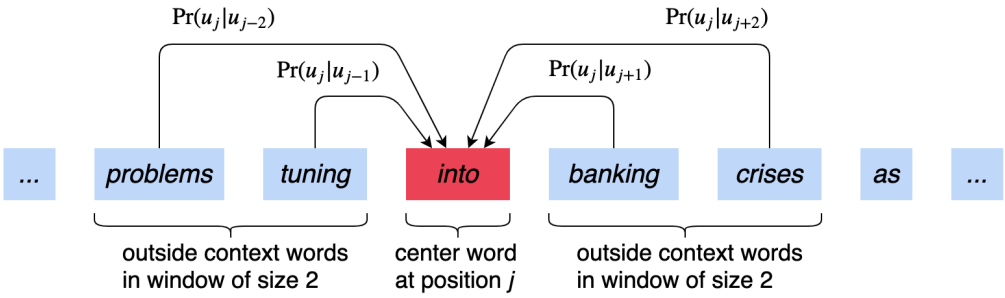


Figure 40: Example windows and process for computing $\Pr(u_j|u_{j+k})$, i.e., CBOW method.

34.1.4 Definition (Not Accomplished)

Skip-gram Model (SG). The original likelihood over corpus $\mathcal{D} = \{\mathcal{U}_1, \cdots, \mathcal{U}_{n_\mathcal{D}}\}$ is shown as follows:

$$\begin{aligned}
\max_{\Theta} \Pr(\mathcal{D};\Theta) &= \prod_{i=1}^{n_\mathcal{D}} \Pr(\mathcal{U}_i;\Theta) \quad \text{(Independence of sequences } \mathcal{U} \text{ in corpus } \mathcal{D}) \\
&= \prod_{i=1}^{n_\mathcal{D}} \prod_{j=1}^{n_{\mathcal{U}_i}} \Pr\left(u_{j-k}^{(i)}, \cdots, u_{j-1}^{(i)}, u_{j+1}^{(i)}, \cdots, u_{j+k}^{(i)} | \, u_j^{(i)}; \Theta\right) \quad \text{(Skip-gram methodology)} \\
&= \prod_{i=1}^{n_\mathcal{D}} \prod_{j=1}^{n_{\mathcal{U}_i}} \prod_{\substack{-k\le\ell\le k \\ \ell\neq 0}} \Pr\left(u_{j+\ell}^{(i)} | u_j^{(i)}; \Theta\right) \quad \text{(Conditional independence assumption among contextual tokens, not good)}
\end{aligned}$$

and the objective $\mathcal{J}(\Theta)$ (to be minimized) is the (average) negative log likelihood:

$$\min_{\Theta} \mathcal{J}(\Theta) = -\frac{1}{n_\mathcal{D}} \sum_{i=1}^{n_\mathcal{D}} \left(\frac{1}{n_{\mathcal{U}_i}} \sum_{i=1}^{n_{\mathcal{U}_i}} \sum_{\substack{-k\le\ell\le k \\ \ell\neq 0}} \log \boxed{\Pr\left(u_{j+\ell}^{(i)} | u_j^{(i)}; \Theta\right)} \right)$$

and the Word2Vec-SG model uses a parameters-shared embedding and softmax regression as the model architecture

$$\Pr\left(u_{j+\ell}^{(i)} | u_j^{(i)}; \Theta\right) = \frac{\exp\left((\boldsymbol{x}_j^{\mathcal{U}_i})^\mathrm{T} \boldsymbol{x}_{j+\ell}^{\mathcal{U}_i}\right)}{\sum_{c=1}^{n_\mathcal{V}} \exp\left((\boldsymbol{x}_j^{\mathcal{U}_i})^\mathrm{T} \boldsymbol{x}_c^{\mathcal{V}}\right)}$$

where $\boldsymbol{x}_j^{\mathcal{U}_i} \in \mathbb{R}^{d_\mathcal{X}}$, $\boldsymbol{x}_{j+\ell}^{\mathcal{U}_i} \in \mathbb{R}^{d_\mathcal{X}}$ and $\boldsymbol{x}_c^{\mathcal{V}} \in \mathbb{R}^{d_\mathcal{X}}$ all come from the embedding lookup operation from token embeddings matrix $\Theta_\mathcal{X} \in \mathbb{R}^{n_\mathcal{V} \times d_\mathcal{X}}$.

The comprehension of above equation to formulate embedding similarity can take three steps:

- $(\boldsymbol{x}_j^{\mathcal{U}_i})^\mathrm{T} \boldsymbol{x}_{j+\ell}^{\mathcal{U}_i} \in \mathbb{R}$: Dot product compares **similarity** between $\boldsymbol{x}_j^{\mathcal{U}_i} \in \mathbb{R}^{d_\mathcal{X}}$ and $\boldsymbol{x}_{j+\ell}^{\mathcal{U}_i} \in \mathbb{R}^{d_\mathcal{X}}$.
- $\exp\left((\boldsymbol{x}_j^{\mathcal{U}_i})^\mathrm{T} \boldsymbol{x}_{j+\ell}^{\mathcal{U}_i}\right) \in \mathbb{R}$: Exponentiation makes anything positive and amplify differences in relative similarity among the center token embedding $\boldsymbol{x}_j^{\mathcal{U}_i}$ and different contextual token embeddings $\boldsymbol{x}_c^{\mathcal{V}}$.
- $\frac{\exp\left((\boldsymbol{x}_j^{\mathcal{U}_i})^\mathrm{T} \boldsymbol{x}_{j+\ell}^{\mathcal{U}_i}\right)}{\sum_{c=1}^{n_\mathcal{V}} \exp\left((\boldsymbol{x}_j^{\mathcal{U}_i})^\mathrm{T} \boldsymbol{x}_c^{\mathcal{V}}\right)}$: Normalize over entire vocabulary $\mathcal{V}$ to give probability distribution.

Indeed, Word2Vec aim to learn similarities between a center token u and any other distinctive tokens $u' \in \mathcal{V}$ through learning the probability distribution $\left(\Pr(u_1^\mathcal{V}|u), \cdots, \Pr(u_{n_\mathcal{V}}^\mathcal{V}|u)\right) \in (0,1)^{n_\mathcal{V}}$.

Continuous Bag-of-Word Model (CBOW). There exists two kinds of CBOW:

- **One-token context CBOW.**
- **Multi-token context CBOW.**

34.1.5 Cognition

Reading Materials.

- Distributed Representations of Words and Phrases and their Compositionality (Mikolov et al., 2013)[58].
- Jay Alammar: The Illustrated Word2vec.
- Word2Vec Parameter Learning Explained (Rong, 2014)[59].
- CS224n: Introduction and Word Vectors.

34.2 Global vectors for word representation (GloVe)

34.2.1 Motivation

34.2.2 Contribution

34.2.3 Definition

34.2.4 Cognition

Reading Materials.

- (HomePage) GloVe: Global Vectors for Word Representation.
- GloVe: Global Vectors for Word Representation (Pennington et al., 2014)[60].

34.3 Deep Contextualized Word Representations (ELMo)

34.3.1 Motivation

- Pre-trained word representations (Mikolov et al., 2013)[58] (Pennington et al., 2014)[60] are a key component in many neural language understanding models.
- Learning high quality representations can be challenging, algorithms should model both:
  1. Complex characteristics of word use (e.g., syntax and semantics).
  2. How these uses vary across linguistic contexts (i.e., to model polysemy).

34.3.2 Contribution

- Introduce a new type of deep contextualized word representation that directly addresses above both challenges.
- Can be easily integrated into existing models.

34.3.3 Definition

Bidirectional Language Models Given a sequence of $n_{\mathcal{U}}$ tokens, $\mathcal{U} = (u_1, u_2, \cdots, u_{n_{\mathcal{U}}})$, where each token index $u \in \{1, \cdots, n_{\mathcal{V}}\}$ and $n_{\mathcal{V}}$ is the vocabulary size of corpus $\mathcal{D}$.

A forward language model (i.e., forward LM) computes the probability of the sequence by modeling the probability of token u_j given the history $u_{j':1\leq j'\leq j-1} = (u_1, \cdots, u_{j-1})$:

$$\begin{aligned}\Pr(\mathcal{U}) &= \prod_{j=2}^{n_{\mathcal{U}}} \Pr(u_j | \, u_{j':1\leq j'\leq j-1}) \\ &= \prod_{j=2}^{n_{\mathcal{U}}} \Pr(u_j | \, u_1, \cdots, u_{j-1})\end{aligned}$$

Lots of neural language models made computation as follows:

1. Compute a **context-independent token representation** $u_j \mapsto \boldsymbol{x}_j^{\text{LM}} \in \mathbb{R}^{d_{\mathcal{X}}}$ (e.g., via token embedding lookup) by vocabulary parameters $\Theta_{\mathcal{X}}$, then pass it through L layers of forward LSTMs with parameters $\overrightarrow{\Theta}_{\text{LSTMs}} = \{\overrightarrow{\Theta}_{\text{LSTMs}}^{(\ell)}\}_{\ell=1}^{L}$.
2. At each token position j inside sequence $\mathcal{U}$, each LSTM layer outputs a **context-dependent encoding representation** $\overrightarrow{\boldsymbol{h}}_{j,\ell}^{\text{LM}} \in \mathbb{R}^{\overrightarrow{d_\ell}}$ of u_j, where $\overrightarrow{d_\ell}$ represents the dimensionality of hidden vector in ℓ-th forward LSTM layer.
3. The top LSTM layer outputs $\overrightarrow{\boldsymbol{h}}_{j,L}^{\text{LM}} \in \mathbb{R}^{\overrightarrow{d_L}}$ at each position $j \in \{1, \cdots, n_{\mathcal{U}}\}$ in sequence $\mathcal{U}$ to predict the next token u_{j+1} following with a softmax layer with linear projection parameters $\Theta_{\mathcal{S}} \in \mathbb{R}^{\overrightarrow{d_L} \times n_{\mathcal{V}}}$.

Similarly, a backward LM runs over the sequence in reverse, predicting the previous token given the future context $u_{j':j+1\leq j'\leq n_{\mathcal{U}}} = (u_{j+1}, \cdots, u_{n_{\mathcal{U}}})$:

$$\begin{aligned}\Pr(\mathcal{U}) &= \prod_{j=1}^{n_{\mathcal{U}}-1} \Pr(u_j | \, u_{j':j+1\leq j'\leq n_{\mathcal{U}}}) \\ &= \prod_{j=1}^{n_{\mathcal{U}}-1} \Pr(u_j | \, u_{j+1}, \cdots, u_{n_{\mathcal{U}}})\end{aligned}$$

Each ℓ-th backward LSTM layer produces representations $\overleftarrow{\boldsymbol{h}}_{j,\ell}^{\text{LM}} \in \mathbb{R}^{\overleftarrow{d_\ell}}$ of u_j given $(u_{j+1}, \cdots, u_{n_{\mathcal{U}}})$, where $\overleftarrow{d_\ell}$ represents the dimensionality of hidden vector in ℓ-th backward LSTM layer.

A biLM combines both a forward and backward LM. This formulation for a specific sequence $\mathcal{U}$ jointly maximizes the log likelihood of the forward and backward directions:

$$\max_{\Theta} \; \mathcal{J}(\Theta) = \sum_{j=2}^{n_{\mathcal{U}}-1} \Big(\log \Pr(u_j | \, u_1, \cdots, u_{j-1}; \Theta_{\mathcal{X}}; \overrightarrow{\Theta}_{\text{LSTMs}}; \Theta_{\mathcal{S}}) + \log \Pr(u_j | \, u_{j+1}, \cdots u_{n_{\mathcal{U}}}; \Theta_{\mathcal{X}}; \overleftarrow{\Theta}_{\text{LSTMs}}; \Theta_{\mathcal{S}}) \Big)$$

where $\Theta = \{\Theta_{\mathcal{X}}, \overrightarrow{\Theta}_{\text{LSTMs}}, \overleftarrow{\Theta}_{\text{LSTMs}}, \Theta_{\mathcal{S}}\}$.

ELMo. A L-layer biLM computes a set of $2L+1$ representations for each token u_j in a training sequence $\mathcal{U}$:

$$\begin{aligned}\mathcal{R}_j &= \left\{ \boldsymbol{x}_j^{\text{LM}}, \overrightarrow{\boldsymbol{h}}_{j,\ell}^{\text{LM}}, \overleftarrow{\boldsymbol{h}}_{j,\ell}^{\text{LM}} \,|\, \ell \in \{1, \cdots, L\} \right\} \\ &= \left\{ \boldsymbol{h}_{j,\ell}^{\text{LM}} \,|\, \ell \in \{0, \cdots, L\} \right\}\end{aligned}$$

where $\boldsymbol{h}_{j,\ell}^{\text{LM}} = \overrightarrow{\boldsymbol{h}}_{j,\ell}^{\text{LM}} \oplus \overleftarrow{\boldsymbol{h}}_{j,\ell}^{\text{LM}} \in \mathbb{R}^{\overrightarrow{d}_\ell + \overleftarrow{d}_\ell}$, $\forall \ell \in \{0, \cdots, L\}$, especially, $\boldsymbol{h}_{j,0}^{\text{LM}}$ is the concatenation of $\boldsymbol{x}_j^{\text{LM}}$ and its inverse.

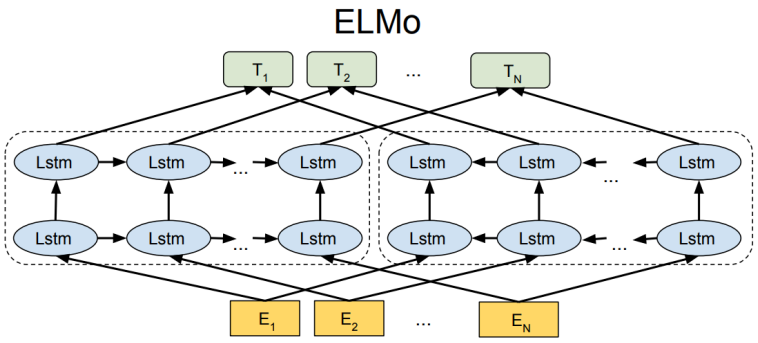


Figure 41: ELMo uses the concatenation of independently trained left-to-right and right-to-left LSTMs to generate features for downstream tasks. Here $\boldsymbol{T}_j = \boldsymbol{h}_{j,L}^{\text{LM}}$ represents the concatenation of the last forward encoding vector and backward encoding vector for a simplest case. Source: (Devlin et al., 2018)[14].

More generally, we compute a task specific weights $\{s_\ell^{\text{task}} \,|\, \ell \in \{1, \cdots, L\}\}$ of all biLM layers:

$$\boldsymbol{T}_j = \gamma^{\text{task}} \sum_{\ell=0}^{L} s_\ell^{\text{task}} \boldsymbol{h}_{j,\ell}^{\text{LM}} \in \mathbb{R}^{\overrightarrow{d}_\ell + \overleftarrow{d}_\ell}$$

where $\boldsymbol{s}^{\text{text}} = (s_1^{\text{text}}, \cdots, s_L^{\text{text}}) \in (0,1)^L$ and $\sum_{\ell=0}^{L} s_\ell^{\text{task}} = 1$ are softmax-normalized weights and $\gamma^{\text{task}} > 0$ is the scalar hyper-parameter that allows the task model to scale the entire ELMo vector.

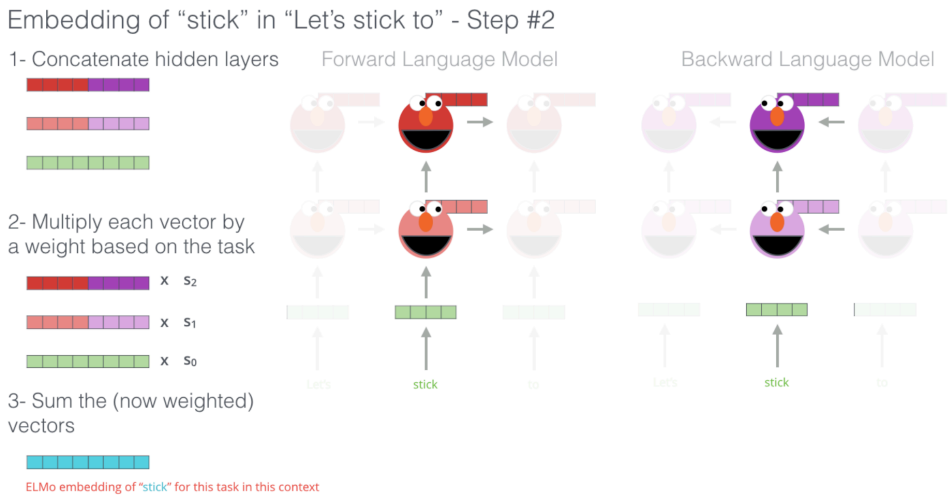


Figure 42: An example for visualizing the $\boldsymbol{T}_j = \gamma^{\text{task}} \sum_{\ell=0}^{L} s_\ell^{\text{task}} \boldsymbol{h}_{j,\ell}^{\text{LM}}$, in this case we compute the embedding of token "stick" in sequence "Let's stick to" with 3 tokens. Source: https://jalammar.github.io/illustrated-bert/.

34.3.4 Cognition

Reading Materials

- Deep Contextualized Word Representations (Peters et al., 2018)[61].
- Jay Alammar: The Illustrated BERT, ELMo, and co. (How NLP Cracked Transfer Learning).

34.4 Improving Language Understanding by Generative Pre-Training (GPT)

34.4.1 Motivation

- Most deep learning methods require substantial amounts of manually labeled data, which restricts their applicability in many domains that suffer from a dearth of annotated resources. Therefore, models that can leverage linguistic information (e.g., token embeddings) from unlabeled data provide a valuable alternative to gathering more annotation, but which can be time-consuming and expensive[15].
- Even in cases where considerable supervision is available, learning good representations in an unsupervised fashion can provide a signicant performance boost (e.g., a semantic similarity assessment uses pre-training word2vec)[15].

34.4.2 Contribution

- Explore a semi-supervised approach for language understanding tasks using a combination of unsupervised pre-training and supervised fine-tuning. Employ a two-stage training procedure:
  1. **Unsupervised pre-training**. Use a language modeling objective on the unlabeled data to learn the initial parameters of a neural network model (task-agnostic).
  2. **Supervised fine-tuning**. Adapt these parameters to a discriminative target task (e.g., document classication) using the corresponding supervised objective (task-specific).
- Using Transformer[11] architecture for handling long-term dependencies among tokens in each text sequence.

34.4.3 Definition

Unsupervised Pre-Training. Given an unsupervised corpus of tokens $\mathcal{D} = \{\mathcal{U}_i = (u_1^{(i)}, \cdots, u_{n_{\mathcal{U}_i}}^{(i)})\}_{i=1}^{n_{\mathcal{D}}}$, where $n_{\mathcal{D}}$ means the total number of sequences (e.g., sentences) in the unsupervised corpus $\mathcal{D}$ and $n_{\mathcal{U}_i}$ means the length of i-th sequence in corpus $\mathcal{D}$. We use a standard language modeling objective to maximize the following likelihood:

$$
\begin{aligned}
\max_{\Theta} \ \mathcal{J}(\Theta) &= \log \Pr(\mathcal{D}; \Theta) = \log \Pr(\{\mathcal{U}_i\}_{i=1}^{n_{\mathcal{D}}}; \Theta) \\
&= \log \Pr\left(\prod_{i=1}^{n_{\mathcal{D}}} \mathcal{U}_i; \Theta\right) \quad (\because \text{i.i.d. of sequences } \mathcal{U}_i \text{ inside corpus } \mathcal{D}) \\
&= \sum_{i=1}^{n_{\mathcal{D}}} \log \Pr(\mathcal{U}_i; \Theta) = \sum_{i=1}^{n_{\mathcal{D}}} \log \Pr\left(\{u_j^{(i)}\}_{j=1}^{n_{\mathcal{U}_i}}; \Theta\right) \\
&= \sum_{i=1}^{n_{\mathcal{D}}} \log \left(\prod_{j=2}^{n_{\mathcal{U}_i}} \Pr\left(u_j^{(i)} | \, u_{j':\max(j-k,1)\le j'\le j-1}^{(i)}; \Theta\right)\right) \quad (k \in (0, n_{\mathcal{U}_i}) \text{ is the contextual window size}) \\
&= \sum_{i=1}^{n_{\mathcal{D}}} \sum_{j=2}^{n_{\mathcal{U}_i}} \log \boxed{\Pr\left(u_j^{(i)} | \, u_{j':\max(j-k,1)\le j'\le j-1}^{(i)}; \Theta\right)}
\end{aligned}
$$

where $\Pr\left(u_j^{(i)} | \, u_{j':\max(j-k,1)\le j'\le j-1}^{(i)}; \Theta\right)$ represents the subset of $\mathcal{U}_i$ which contains all tokens stand before $u_j^{(i)}$ and after $u_{j-k}^{(i)}$ simultaneously if $u_{j-k}^{(i)}$ existing. Namely, if if $u_{j-k}^{(i)}$ exists, there exists the following equivalent set notation:

$$
u_{j':\max(j-k,1)\le j'\le j-1}^{(i)} = (u_{j-k}^{(i)}, \cdots, u_{j-1}^{(i)})
$$

Each conditional probability $\Pr(u_j^{(i)} | \, u_{j':\max(j-k;1)\le j'\le j-1}^{(i)}; \Theta)$ is modeled using a neural network with parameters Θ. These parameters can be trained using stochastic gradient descent[23] and so on. In GPT paper, they use **a multi-layer Transformer decoder without encoder-decoder attention** (i.e., Transformer decoder-only block, masked multi-head attention and feed forward network, see Figure 43) for the language model, which is a variant of the Transformer.

The practical computation process shows as follows:

$$
\begin{aligned}
\boldsymbol{H}_{\mathcal{U}}^{(0)} &= \boldsymbol{U}\boldsymbol{\Theta}_{\mathcal{X}} + \boldsymbol{P}_{\mathcal{U}} \in \mathbb{R}^{n_{\mathcal{U}} \times d_{\mathcal{X}}} \quad (\mathcal{U} \in \mathbb{Z}^{n_{\mathcal{U}}} \mapsto \boldsymbol{U} \in \{0,1\}^{n_{\mathcal{U}} \times n_{\mathcal{V}}} \text{ (one-hot encodings)}) \\
\boldsymbol{H}_{\mathcal{U}}^{(\ell)} &= \text{Transformer\_Decoder\_Block}(\boldsymbol{H}_{\mathcal{U}}^{(\ell-1)}) \in \mathbb{R}^{n_{\mathcal{U}} \times d_{\mathcal{X}}}, \ \forall \ell \in \{1, \cdots, L\} \\
\Pr(\mathcal{U}) &= \text{softmax}(\boldsymbol{H}_{\mathcal{U}}^{(L)} \boldsymbol{\Theta}_{\mathcal{X}}^{\mathrm{T}}) \in \mathbb{R}^{n_{\mathcal{U}} \times n_{\mathcal{V}}}
\end{aligned}
$$

where:

- $\boldsymbol{\Theta}_{\mathcal{X}} \in \mathbb{R}^{n_{\mathcal{V}} \times d_{\mathcal{X}}}$: Embeddings for all unique tokens within the corpus $\mathcal{D}$. $n_{\mathcal{V}}$ is the number of unique tokens inside $\mathcal{D}$ (i.e., vocabulary size of corpus).
- $\boldsymbol{U}\boldsymbol{\Theta}_{\mathcal{X}} \in \mathbb{R}^{n_{\mathcal{U}} \times d_{\mathcal{X}}}$: Embeddings lookup for the sequence $\mathcal{U}$, $n_{\mathcal{U}}$ is the length (number of tokens) of $\mathcal{U}$.
- $\boldsymbol{P}_{\mathcal{U}} \in \mathbb{R}^{n_{\mathcal{U}} \times d_{\mathcal{X}}}$: Positional embedding matrix.
- $\boldsymbol{H}_{\mathcal{U}}^{(L)} \in \mathbb{R}^{n_{\mathcal{U}} \times d_{\mathcal{X}}}$: Final encoding vectors of all tokens inside the sequence $\mathcal{U}$, containing long-dependencies information.
- $\Pr(\mathcal{U}) \in \mathbb{R}^{n_{\mathcal{U}} \times n_{\mathcal{V}}}$: Probability distributions over vocabulary of token corresponding all input tokens of sequence $\mathcal{U}$.

Note here:

- Transformer_Decoder_Block doesn't contain encoder-decoder module, only contains masked multi-head attention module and feed forward network module. See Figure 43.
- Because we have specific label token u_j for each position j, we can impose parallel softmax computing for all positions using the whole encoding matrix $\boldsymbol{H}_{\mathcal{U}}^{(L)}$, and get the whole token probability distributions corresponding all tokens inside sequence $\Pr(\mathcal{U})$.

After unsupervised pre-training, we have the final parameters

$$\Theta = \{\boldsymbol{\Theta}_{\mathcal{X}}\} \cup \{\boldsymbol{\Theta}_{\mathcal{P}}\} \cup \underbrace{\{\boldsymbol{W}_{\mathrm{Q}}^{(\ell,h)}, \boldsymbol{W}_{\mathrm{K}}^{(\ell,h)}, \boldsymbol{W}_{\mathrm{V}}^{(\ell,h)}, \boldsymbol{W}_{\mathrm{O}}^{(\ell)} | \, \ell \in \{1, \cdots, L\}, h \in \{1, \cdots, H_\ell\}\}}_{\text{masked multi-head attention}}$$
$$\cup \underbrace{\{\boldsymbol{W}_{\text{ffn}-1}^{(\ell)}, \boldsymbol{W}_{\text{ffn}-2}^{(\ell)}, \boldsymbol{b}_{\text{ffn}-1}^{(\ell)}, \boldsymbol{b}_{\text{ffn}-2}^{(\ell)} | \, \ell \in \{1, \cdots, L\}\}}_{\text{feed forward}}$$

where H_ℓ is the number of self-attention heads in ℓ-th Transformer block.

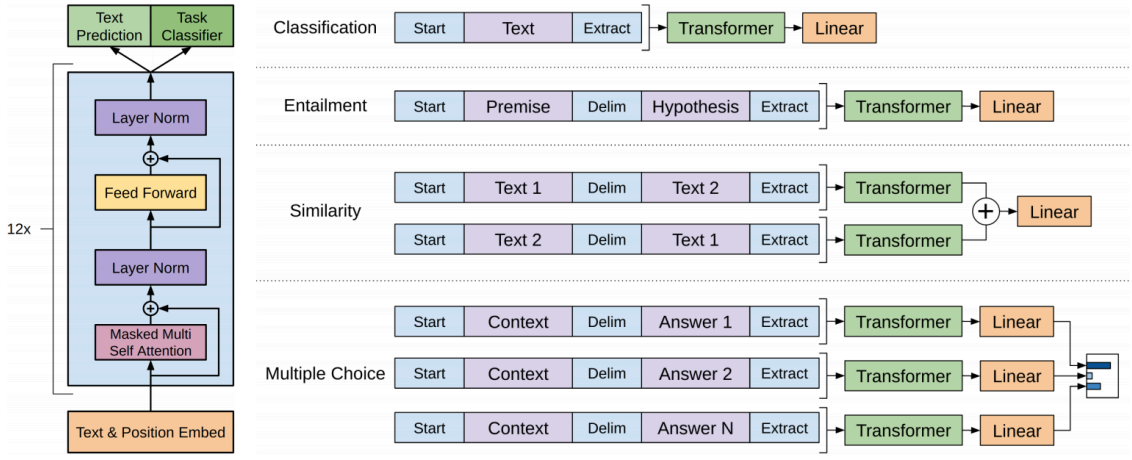


Figure 43: (left) Transformer architecture and training objectives used in this work. (right) Input transformations for fine-tuning on different tasks. We convert all structured inputs into token sequences to be processed by our pre-trained model, followed by a linear+softmax layer. Source: (Radford et al., 2018)[15].

Supervised Fine-Tuning. After training the model with above objective, we adapt the parameters to the supervised target task. For each supervised training sample $(\mathcal{U}, Y)$:

- Each input sequence $\mathcal{U} \in \mathbb{R}^{n_{\mathcal{U}}}$ is passed through our pre-trained model to obtain the final Transformer blocks activation $\boldsymbol{H}_{\mathcal{U}}^{(L)} \in \mathbb{R}^{n_{\mathcal{U}} \times d_{\mathcal{X}}}$.
- Then feed $\boldsymbol{H}_{\mathcal{U}}^{(L)}$ into an added linear projection layer with parameters $\Theta_{\mathcal{S}}$ to predict Y (e.g., for token classification task, $\Theta_{\mathcal{S}} \in \mathbb{R}^{d_{\mathcal{X}} \times n_C}$ and $Y \sim \mathbb{R}^{n_{\mathcal{U}} \times n_C}$).

34.5 Language Models are Unsupervised Multitask Learners (GPT-2)

34.5.1 Motivation

- Combination of pre-training and supervised fine-tuning is the trend of more general methods of transfer in NLP.

34.5.2 Contribution

- Combination of pre-training and supervised fine-tuning.
- Learning from more domain corpuses.

34.5.3 Definition

Architecture of Transformer-Decoder-Only Module in GPT-2. Architecture of OpenAI GPT-2 model shows as follows, it still stacks multi Transformer Decoder-only blocks. The differences between GPT-2 and GPT are shown as

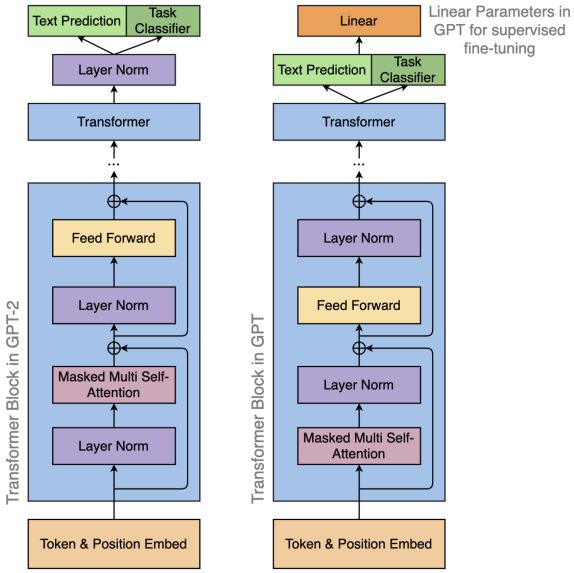


Figure 44: (left) Architecture of OpenAI GPT-2 (Ralford et al., 2018a)[16]. (right) OpenAI GPT (Ralford et al., 2018)[15] model.

follows:

- There doesn't exist fine-tuning layer anymore. GPT-2 will automatically identify what tasks is going to be done.
- More training dataset, about 40G high-quality corpus.
- More Transformer blocks (i.e., $L = 48$ layers), deeper embedding size $d_{\mathcal{X}} = 1600$, exist $n_{\Theta} = 1.5$ billions total parameters (0.3 billion in BERT).
- Change the order of masked multi-head attention and layer norm, feed forward and layer norm.
- Vocabulary size $n_{\mathcal{V}}$ increases to 50257. Context size $n_{\mathcal{U}}$ increases from 512 tokens to 1024 tokens. Batch size $n_{\mathcal{B}}$ increases to 512.

34.5.4 Cognition

Reading Materials.

- Jay Alammar: The Illustrated GPT-2 (Visualizing Transformer Language Models).
- (OpenAI GPT) Improving Language Understanding by Generative Pre-Training (Ralford et al., 2018)[15].
- (OpenAI GPT-2) Language Models are Unsupervised Multitask Learners (Ralford et al., 2018a)[16].

34.6 Pre-training of Deep Bidirectional Encoder Representation from Transformers (BERT)

34.6.1 Motivation

- ELMo (Peters et al., 2018)[61] indeed is not a bidirectional language model, Only output a concatenation of an encoding vector of forward language model and an encoding vector of backward language model.
- OpenAI GPT (Ralford et al., 2018)[15] only use a left-to-right architecture, where every token can only attend to previous tokens in the self-attention layers of the Transformer (Vaswani et al., 2017)[11]. Such restrictions are sub-optimal for sentence-level tasks, and could be very harmful when applying fine-tuning based approaches to token-level tasks such as question answering, where it is crucial to incorporate context from both directions.

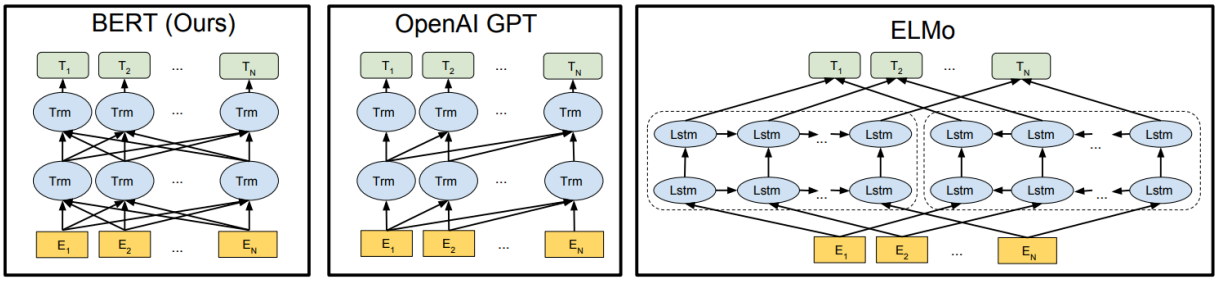


Figure 45: Differences in pre-training model architectures. BERT uses a bidirectional Transformer (i.e., Transformer encoder-only blocks). OpenAI GPT uses a left-to-right Transformer (i.e., Transformer decoder-only blocks). ELMo uses the concatenation of independently trained left-to-right and right-toleft LSTMs to generate features for downstream tasks. Among the three, only BERT representations are jointly conditioned on both left and right context in all layers. In addition to the architecture differences, **BERT and OpenAI GPT are fine-tuning approaches, while ELMo is a feature-based approach**. Source: (Devlin et al., 2018)[14].

34.6.2 Contribution

- Demonstrate the importance of bidirectional pre-training for language representations.
- Use masked language models to enable pre-trained deep bidirectional representations.
- The first fine-tuning based representation model that achieves state-of-the-art performance on a large suite of sentence-level and token-level tasks, outperforming many task-specic architectures.

34.6.3 Definition

There are two steps in BERT framework: pre-training & fine-tuning.

- Pre-training: the BERT model is trained on unlabeled data over different pre-training tasks.
- Fine-tuning: the BERT model is initialized with the pre-trained parameters, and all of the parameters are fine-tuned using labeled data from the downstream tasks. Each downstream task has separate ne-tuned models, even though they are initialized with the same pre-trained parameters.

Model Architecture. Using Transformer encoder block (i.e., see Figure 28) (GPT using Transformer decoder-only block) as the base block in BERT. Primarily report results on two model sizes:

- $\text{BERT}_{\text{BASE}}(\text{Num Blocks} = 12, \text{Hidden Size} = 768, \text{Num Heads} = 12, \text{Total Parameters} = 110\text{M})$
- $\text{BERT}_{\text{LARGE}}(\text{Num Blocks} = 24, \text{Hidden Size} = 1024, \text{Num Heads} = 12, \text{Total Parameters} = 340\text{M})$

Input/Output Representations. In BERT, a input sequence of tokens may contain multi sentences (e.g., $\mathcal{U} =< \mathcal{U}_{\text{Q}}, \mathcal{U}_{\text{A}} >$ in question answering task.)

And for a given token u, its input representation is constructed by summing the corresponding token, segmentation, and position embeddings. A visualization of this construction can be seen in Figure 46. Note we use a special classification

token [CLS] as the first token of every sequence, and use another special classification token [SEP] to seperate different segmentations (e.g., $\mathcal{U}_{\mathrm{Q}}$ and $\mathcal{U}_{\mathrm{A}}$) inside a sequence $\mathcal{U}$.

As the output of BERT, each token u (also [CLS] and [SEP]) inside the sequence is represented as a encoding vector $\boldsymbol{r} \in \mathbb{R}^{d_{\mathcal{X}}}$ (i.e., $\boldsymbol{C}$, $\boldsymbol{T}_j$, $\boldsymbol{T}_{\mathrm{SEP}}$ and $\boldsymbol{T}'_j$ in Figure 47).

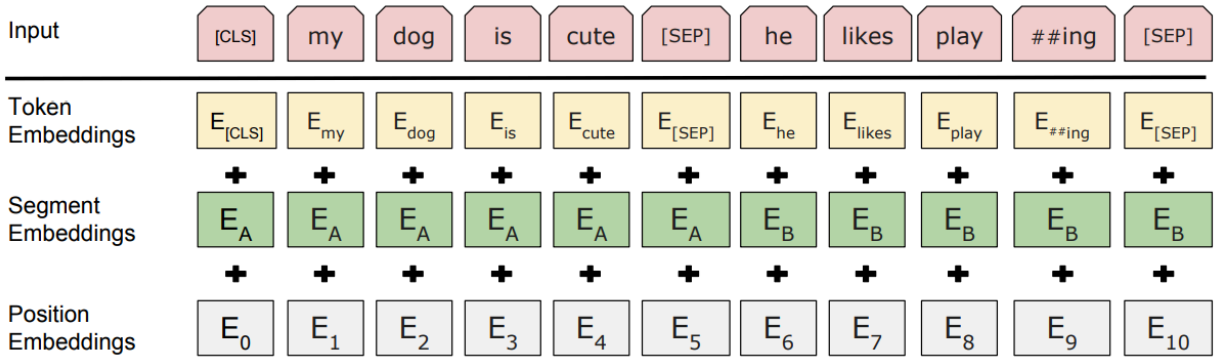


Figure 46: BERT input representation. The input embeddings are the sum of the token embeddings, the segmentation embeddings and the position embeddings. Source: (Devlin et al., 2018)[14].

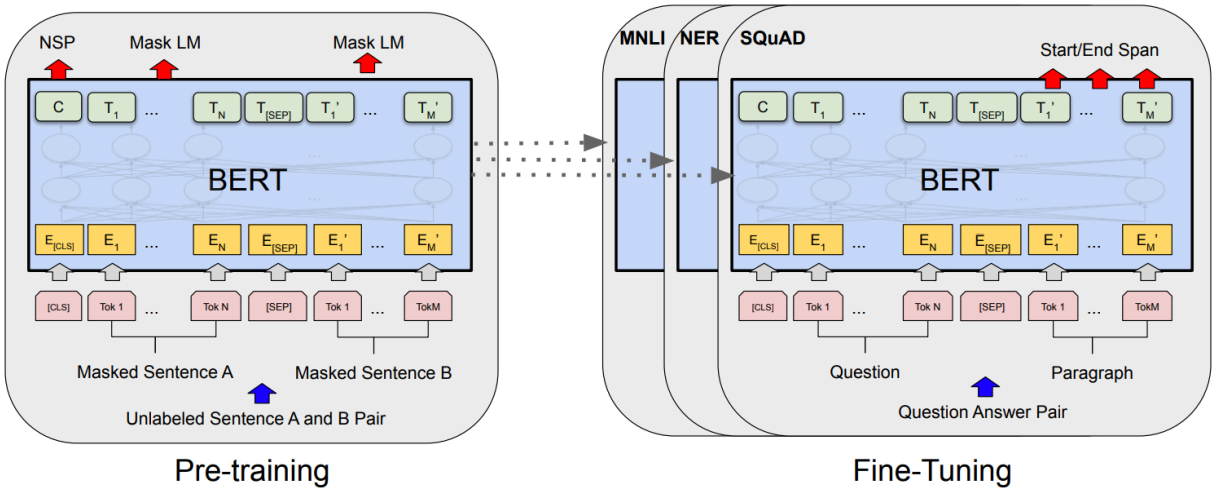


Figure 47: Overall pre-training and fine-tuning procedures for BERT. Apart from output layers, the same architectures are used in both pre-training and fine-tuning. The same pre-trained model parameters are used to initialize models for different down-stream tasks. During fine-tuning, all parameters are fine-tuned. [CLS] is a special symbol added in front of every input example, and [SEP] is a special separator token (e.g. separating questions/answers). Source: (Devlin et al., 2018)[14].

Pre-training BERT. Unlike (Peters et al., 2018)[61]:

$$\max_{\Theta} \mathcal{J}(\Theta; \mathcal{U}) = \sum_{j=2}^{n_{\mathcal{U}}-1} \left(\log \Pr(u_j | u_1, \cdots, u_{j-1}; \Theta_{\mathcal{X}}; \overrightarrow{\Theta}_{\mathrm{LSTMs}}; \Theta_{\mathcal{S}}) + \log \Pr(u_j | u_{j+1}, \cdots u_{n_{\mathcal{U}}}; \Theta_{\mathcal{X}}; \overleftarrow{\Theta}_{\mathrm{LSTMs}}; \Theta_{\mathcal{S}}) \right)$$

where $\Theta = \{\Theta_{\mathcal{X}}, \overrightarrow{\Theta}_{\mathrm{LSTMs}}, \overleftarrow{\Theta}_{\mathrm{LSTMs}}, \Theta_{\mathcal{S}}\}$, and (Ralford et al., 2018)[15]:

$$\max_{\Theta} \mathcal{J}(\Theta; \mathcal{D}) = \sum_{i=1}^{n_{\mathcal{D}}} \sum_{j=2}^{n_{\mathcal{U}_i}} \log \Pr \left(u_j^{(i)} | u_{j':\max(j-k;1) \le j' \le j-1}^{(i)}; \Theta \right)$$

where

$$\Theta = \{\boldsymbol{\Theta}_{\mathcal{X}}\} \cup \{\boldsymbol{\Theta}_{\mathcal{P}}\} \cup \underbrace{\{\boldsymbol{W}_{\mathrm{Q}}^{(\ell,h)}, \boldsymbol{W}_{\mathrm{K}}^{(\ell,h)}, \boldsymbol{W}_{\mathrm{V}}^{(\ell,h)}, \boldsymbol{W}_{\mathrm{O}}^{(\ell)} | \ell \in \{1, \cdots, L\}, h \in \{1, \cdots, H_\ell\}\}}_{\text{masked multi-head attention}}$$

$$\cup \underbrace{\{\boldsymbol{W}_{\text{ffn}-1}^{(\ell)}, \boldsymbol{W}_{\text{ffn}-2}^{(\ell)}, \boldsymbol{b}_{\text{ffn}-1}^{(\ell)}, \boldsymbol{b}_{\text{ffn}-2}^{(\ell)} | \, \ell \in \{1, \cdots, L\}\}}_{\text{feed forward}}$$

Authors didn't use traditional left-to-right or right-to-left language models to pre-train BERT. Instead, a bidirectional alternative as follows:

$$\max_{\Theta} \ \mathcal{J}(\Theta) = \sum_{i=1}^{n_{\mathcal{D}}} \sum\nolimits_{u_j^{(i)} \in \mathcal{U}_i} \log \Pr\left(u_j^{(i)} | \, u_1^{(i)}, \cdots, u_{j-1}^{(i)}, u_{j+1}^{(i)}, \cdots, u_{n_{\mathcal{U}_i}}^{(i)}; \Theta\right)$$

They trained BERT by two unsupervised tasks as follows:

- **Masked Language Modeling (MLM)**. A multi classification (e.g., follow a linear projection layer $\Theta_{\mathcal{S}} \in \mathbb{R}^{d_{\mathcal{X}} \times n_{\mathcal{V}}}$ and perform softmax operation after Transformer blocks) over the whole vocabulary of corpus. Simply mask some percentage of the input tokens at random, and then predict these masked tokens.

  Assuming the unlabeled sentence is "my dog is hairy", and during the random masking procedure we chose to predict the 4-th token (i.e., "hairy" as the ground-truth label), the masking procedure can be further illustrated by:
  - 80% of the time: Replace the word with the [MASK] token, e.g., "my dog is hairy" $\longrightarrow$ "my dog is [MASK]" as the current practical input sequence.
  - 10% of the time: Replace the word with a random word, e.g., "my dog is hairy" $\longrightarrow$ "my dog is apple" as the current practical input sequence. The purpose of this is to prevent overfitting over corpus.
  - 10% of the time: Keep the word unchanged, e.g., "my dog is hairy" $\longrightarrow$ "my dog is hairy" as the current practical input sequence. The purpose of this is to bias the representation towards the actual observed word, i.e., to learn the influence of token u_j on self encoding $\boldsymbol{r}_j$: $u_j \rightarrow \boldsymbol{r}_j$.
- **Next Sentence Prediction (NSP)**. A binary classification (e.g., follow a linear layer $\Theta_{\mathcal{S}} \in \mathbb{R}^{d_{\mathcal{X}} \times 1}$ and perform logistic operation after Transformer blocks) with input feature vector $\boldsymbol{r}_0 \in \mathbb{R}^{d_{\mathcal{X}}}$ (i.e., $\boldsymbol{C} \in \mathbb{R}^{d_{\mathcal{X}}}$ in Figure 47, because it encodes all dependences of tokens inside the whole sequence). e.g., sequences and corresponding labels as follows:
  - Input: "[CLS] the man went to [MASK] store [SEP] he bought a gallon [MASK] milk [SEP]"
    Label: IsNext
  - Input: "[CLS] the man [MASK] to the store [SEP] penguin [MASK] are flight less birds [SEP]"
    Label: NotNext

The detailed pre-training procedure can be found in appendix part of (BERT, Devlin et al., 2018)[14].

Fine-tuning BERT. For each task, simply plug in the task-specific inputs and outputs (e.g., in QA task, input segmentation and corresponding output segmentation are concated in the same sequence, separated by a special token [SEP]) into BERT and fine-tuning all parameters end-to-end.

Compared to pre-training, fine-tuning is relatively inexpensive, can be replicated in at most 1 hour one the single Cloud TPU, or a few hours on a GPU, starting from the pre-trained model.

The illustration of fine-tuning BERT on different tasks can be seen in Figure 48. Task-specific BERT models are formed by incorporating BERT with one additional output layer, so a minimal number of parameters need to be learned from scratch. Among the tasks, (a) and (b) are sequence-level tasks while (c) and (d) are token-level tasks. In Figure 48, $\boldsymbol{E}$ represents the input embedding, $\boldsymbol{T}_i$ represents the contextual representation of token u_i, [CLS] is the special symbol for classification output, and [SEP] is the special symbol to separate non-consecutive token sequences (i.e., different segmentations).

The other detailed fine-tuning procedure can be found in appendix part of (BERT, Devlin et al., 2018)[14].

34.6.4 Continuation

RoBERTa. (Liu et al., 2019)[62] present a replication study of BERT pretraining that carefully measures the impact of many key hyperparameters and training data size. Details see paper RoBERTa: A Robustly Optimized BERT Pretraining Approach.

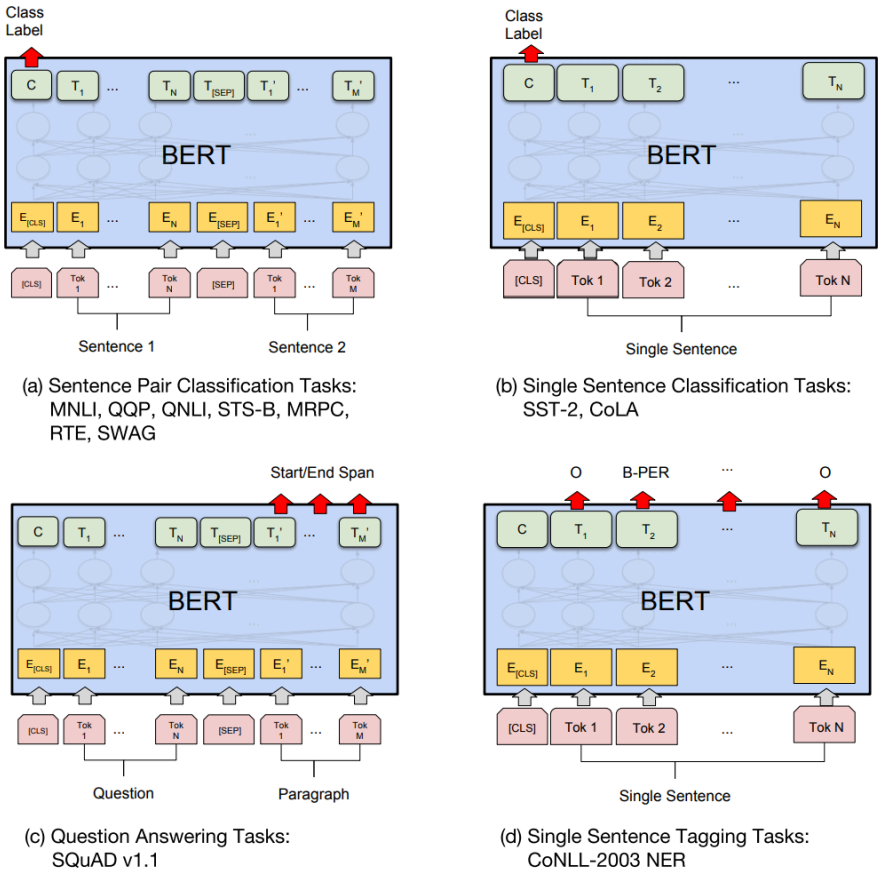
34.6.5 Cognition

Opinions.

- During the fine-tuning phase, the official suggests to fine-tune all of the parameters of BERT using labeled data, but from the perspective of saving costs and preventing overfitting, we can try to fix the underlying parameters and only train the added last task-specific linear layer.
- Adding piror information (e.g., a piror token distribution) inside random masking procedure during pre-training BERT.

Reading Materials.

- BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding (Devlin et al., 2018)[14].
- Jay Alammar: The Illustrated BERT, ELMo, and co. (How NLP Cracked Transfer Learning).
- (Slides Jacob Devlin, Google AI Language) BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding .
- RoBERTa: A Robustly Optimized BERT Pretraining Approach (Liu et al., 2019)[62].
- A Primer in BERTology: What we know about how BERT works (Rogers et al., 2020.02).



(a) Sentence Pair Classification Tasks: MNLI, QQP, QNLI, STS-B, MRPC, RTE, SWAG

(b) Single Sentence Classification Tasks: SST-2, CoLA

(c) Question Answering Tasks: SQuAD v1.1

(d) Single Sentence Tagging Tasks: CoNLL-2003 NER

Figure 48: Illustrations of Fine-tuning BERT on Different Tasks. Source: (Devlin et al., 2018)[14].

34.7 Improving Pre-training by Representing and Predicting Spans (SpanBERT)

34.7.1 Motivation

- Many NLP tasks involve reasoning about relationships between two or more spans of text.

34.7.2 Contribution

- SpanBERT (Joshi et al., 2019)[17] extends BERT (Devlin et al., 2018)[14] by:
  1. Masking contiguous random spans, rather than random tokens.
  2. Training with the span boundary representations to predict the entire content of the masked span, without relying on the individual token representations within the sequence.

34.7.3 Definition

Pre-training SpanBERT. The model architecture of SpanBERT (Joshi et al., 2019)[17] is same as BERT (Devlin et al., 2018)[14], stacking multi Transformer (Vaswani et al., 2017)[11] encoder blocks (see Figure 28). The only differences between SpanBERT and BERT are using different approachs (i.e., self-supervised tasks) to pre-train the model. BERT uses MLM and NSP tasks (see in Subsubsection 34.6.3), but SpanBERT uses the following three tasks:

- **Span Masking**.

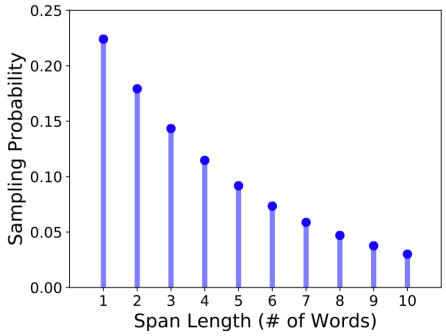


Figure 49: SpanBERT sample random span lengths from a geometric distribution $l \sim \text{Geo}(p = 0.2)$ clipped at $l_{\max} = 10$. which is skewed towards shorter spans. This yields a mean span length of $\text{mean}(l) = 3.8$. Source: (Joshi et al., 2019)[17].

  Same as BERT, SpanBERT also mask 15% of tokens in total: replacing 80% of the masked tokens with [MASK] (to be set score as $-\inf$ in self-attention operation later), 10% with random tokens and 10% with the original tokens. Different from BERT, SpanBERT replace all tokens in a span, not for each token. i.e., all the tokens in a span are replaced with [MASK] or sampled tokens.

- **Span Boundary Objective (SBO)**. Predict the entire masked span using only the representations of the tokens at the span's boundary.

  Formally, denote the output encoding vector from Transformer encoder for each token in the sequence by $\boldsymbol{r}_1^{\mathcal{U}}, \cdots, \boldsymbol{r}_{n_{\mathcal{U}}}^{\mathcal{U}} \in \mathbb{R}^{d_{\mathcal{X}}}$. Given a masked span of tokens $(u_s, \cdots, u_e) \in \mathcal{U}$, where (s, e) indicates its start and end position. We represent each token u_j in the span using the output encodings of the external boundary tokens $\boldsymbol{r}_{s-1}^{\mathcal{U}}$ and $\boldsymbol{r}_{e+1}^{\mathcal{U}}$ as well as the position embedding of the target token $\boldsymbol{p}_{j-s+1}^{\mathcal{U}}$:

$$\boldsymbol{y}_j^{\mathcal{U}} = f(\boldsymbol{r}_{s-1}^{\mathcal{U}}, \boldsymbol{r}_{e+1}^{\mathcal{U}}, \boldsymbol{p}_{j-s+1}^{\mathcal{U}})$$

  where position embeddings $\boldsymbol{p}_1, \boldsymbol{p}_2, \cdots$ mark **relative positions of the marked tokens with respect to the left boundary token** u_{s-1}. $f(\cdot)$ is added layers after Transformer blocks.

  In SpanBERT paper, authors implement $f(\cdot)$ here as a 2-layer feed forward network with GeLUs activations (Hendrycks and Gimpel, 2016)[63] and layer normalization (Bai et al., 2016)[41]:

$$\begin{aligned}
\boldsymbol{h}_{i,0}^{\mathcal{U}} &= \boldsymbol{r}_{s-1}^{\mathcal{U}} \oplus \boldsymbol{r}_{e+1}^{\mathcal{U}} \oplus \boldsymbol{p}_{i-s+1}^{\mathcal{U}} \\
\boldsymbol{h}_{i,1}^{\mathcal{U}} &= \text{LayerNorm}(\text{GeLU}(\boldsymbol{h}_{i,0}^{\mathcal{U}} \boldsymbol{W}_1)) \\
\boldsymbol{y}_i^{\mathcal{U}} &= \text{LayerNorm}(\text{GeLU}(\boldsymbol{h}_{i,1}^{\mathcal{U}} \boldsymbol{W}_2))
\end{aligned}$$

We then use the vector representation $\boldsymbol{y}_i$ to predict the token u_i and compute the cross-entropy loss exactly like the MLM objective.

SpanBERT sums the loss from both the span boundary and the regular masked language model objectives for each token u_i in the masked span (u_s, u_e), while reusing the input embedding for the target tokens in both MLM and SBO:

$$\begin{aligned}\mathcal{J}(\Theta; u_i) &= \mathcal{J}_{\text{MLM}}(\Theta; u_i) + \mathcal{J}_{\text{SBO}}(\Theta; u_i) \\ &= \log \Pr(u_i | \boldsymbol{x}_i^{\mathcal{U}}; \Theta) + \log \Pr(u_i | \boldsymbol{y}_i^{\mathcal{U}}; \Theta)\end{aligned}$$

- **Single-Sequence Training**. Because SpanBERT doesn't use BERT's NSP objective, only sample a single contiguous segment of the tokens for a training sequence (instead of two segments as a sequence).

$$\begin{aligned}\mathcal{L}(\text{football}) &= \mathcal{L}_{\text{MLM}}(\text{football}) + \mathcal{L}_{\text{SBO}}(\text{football}) \\ &= -\log P(\text{football} \mid \mathbf{x}_7) - \log P(\text{football} \mid \mathbf{x}_4, \mathbf{x}_9, \mathbf{p}_3)\end{aligned}$$

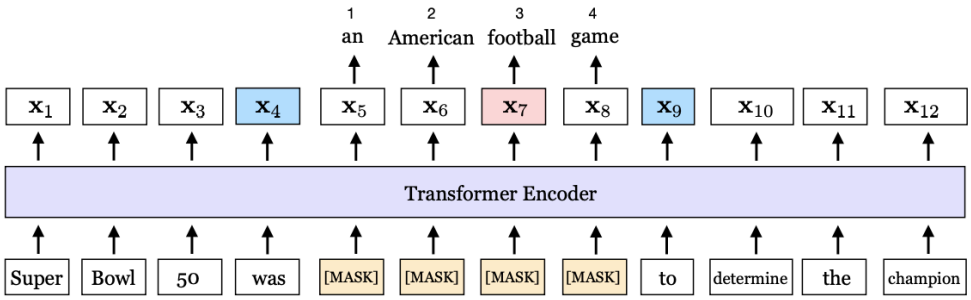


Figure 50: An illustration of SpanBERT training. The span "an American football game" is masked. The span boundary objective (SBO) uses the output representations of the boundary tokens $\boldsymbol{x}_4 \in \mathbb{R}^{d_x}$ and $\boldsymbol{x}_9 \in \mathbb{R}^{d_x}$ (in blue), to predict each token in the masked span. The equation shows the MLM and SBO objective terms for predict the token "football" (in pink), which as marked by the position embedding $\boldsymbol{p}_3 \in \mathbb{R}^{d_x}$, is the 3-th token from $u_4 \in \{1, \cdots, n_{\mathcal{V}}\}$. Source: (Joshi et al., 2019)[17].

34.7.4 Cognition

Opinions.

- Another computing method $f(\cdot)$ for token embedding and relative position embedding (i.e., not sum operation).
- Consider not only $\boldsymbol{p}_{i-s+1}^{\mathcal{U}}$ but also $\boldsymbol{p}_{e-i+1}^{\mathcal{U}}$.

Reading Materials.

- SpanBERT: Improving Pre-training by Representing and Predicting Spans (Joshi et al., 2019)[17].
- Zhihu: Talking about Knowledge-Injected BERTs.

34.8 Unified Language Model Pre-training for Natural Language Understanding and Generation (UniLM)

34.8.1 Motivation

- BERT (Devlin et al., 2018)[14] can be only used in NLU (nature language understanding) tasks, not NLG (nature language generation) tasks, because it uses bidirectional LM, not unidirectional LM.
- ELMo (Peters et al., 2018)[61] and GPT (Ralford et al., 2018)[15] have worse performance than BERT (Devlin et al., 2018)[14] in NLG because they only use unidirectional LM, not bidirectional LM.

Table 4: Comparison between language model (LM) pre-training objectives.

	ELMo	GPT	BERT	UniLM
Left-to-Right LM	✓	✓		✓
Right-to-Left LM	✓			✓
Bidirectional LM			✓	✓
Sequence-to-Sequence LM				✓

34.8.2 Contribution

- Present a new Unified pre-trained Language Model (UniLM) that can be fine-tuned for both natural language understanding and generation tasks.

34.8.3 Definition

Pre-training UniLM. (Dong et al., 2019)[18] pretrain UniLM using four cloze tasks designed for different language modeling objectives. In a cloze task, authors randomly choose some WordPiece tokens in the input, and replace them with special token [MASK]. Then, authors feed their corresponding output encoding vectors computed by the Transformer network into a softmax classifier to predict the masked token. The parameters of UniLM are learned to minimize the cross-entropy loss computed using the predicted tokens and the original tokens.

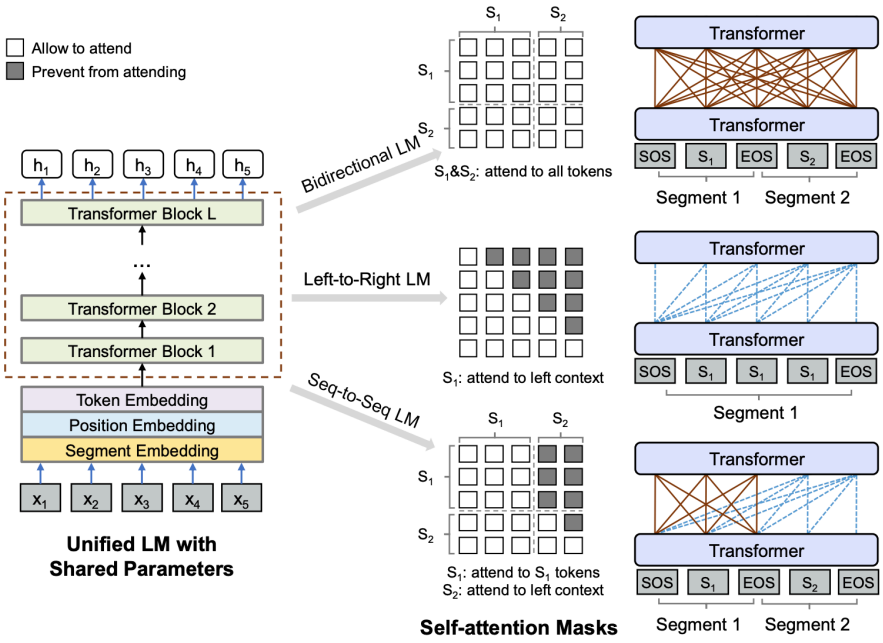


Figure 51: Overview of unified LM pre-training. The model parameters are shared across the LM objectives (i.e., bidirectional LM, unidirectional LM, and sequence-to-sequence LM). We use different self-attention masks to control the access to context for each word token. The right-to-left LM is similar to the left-to-right one, which is omitted in the figure for brevity. Source: (Dong et al., 2019)[18].

- **Unidirectional LM.** Use both left-to-right and right-to-left LM objectives. e.g., a left-to-right LM example, the representation of each token encodes only the leftward context tokens and itself. For instance, to predict the masked token of $< u_1, u_2, [\text{MASK}], u_4 >$, only tokens u_1, u_2 and itself can be used.
  This is done by using a triangular matrix for the self-attention mask $\boldsymbol{M} \in \mathbb{R}^{n_{\mathcal{U}} \times n_{\mathcal{U}}}$, where $n_{\mathcal{U}}$ is the length of sequence (e.g., $n_{\mathcal{U}} = 4$ here). The upper triangular part of the self-attention mask is set to $-\inf$, and the other elements to 0, as shown in Figure 51. Similarly, a right-to-left LM predicts a token conditioned on its future (right) context.
- **Bidirectional LM.** Allows all tokens to attend to each other in prediction. It encodes contextual information from both directions, and can generate better contextual representations of text than its unidirectional counterpart.
  The self-attention mask $\boldsymbol{M} \in \mathbb{R}^{n_{\mathcal{U}} \times n_{\mathcal{U}}}$ is a zero matrix, so that every token is allowed to attend across all positions in the input sequence $\mathcal{U} = (u_1, \cdots, u_{n_{\mathcal{U}}})$.
- **Sequence-to-Sequence LM.** As shown in Figure 51, for prediction, the tokens in the first (source) segment can attend to each other from both directions within the segment, while the tokens of the second (target) segment can only attend to the leftward context in the target segment and itself, as well as all the tokens in the source segment.
  For example, given source segment "u_1, u_2" and its target segment "u_3, u_4, u_5", we feed input $\mathcal{U}$ = <[SOS] u_1 u_2 [EOS] u_3 u_4 u_5 [EOS]> into the model. While both u_1 and u_2 have access to the first four tokens, including [SOS] and [EOS], u_4 can only attend to the first six tokens (i.e., [SOS], u_1, u_2, [EOS], u_3 and u_4).
  Figure 51 shows the self-attention mask $\boldsymbol{M}$ used for the sequence-to-sequence LM objective.
  - The left part of $\boldsymbol{M}$ is set to 0 so that all tokens can attend to the first segment.
  - The upper right part of $\boldsymbol{M}$ is set to $-\inf$ to block attentions from the source segment to the target segment.
  - The lower right part of $\boldsymbol{M}$, we set its upper triangular part to $-\inf$ and the other elements to 0, which prevents tokens in the target segment from attending their future (right) positions.

  During training, we randomly choose tokens in both segments, and replace them with the special token [MASK]. In self-attention mask $\boldsymbol{M}$

$$u_j = [\text{mask}] \mapsto \mathrm{M}_{ij} = -\inf, \forall i \in \{1, \cdots, n_{\mathcal{U}}\}$$

The overall training objective is the sum of different types of LM objectives described above. Specifically, within one training batch, 1/3 of the time we use the bidirectional LM objective, 1/3 of the time we employ the sequence-to-sequence LM objective, and both left-to-right and right-to-left LM objectives are sampled with rate of 1/6.

Fine-tuning UniLM.

- **NLU task.** Use the encoding vector of [SOS] as the representation of input, denoted as $\boldsymbol{h}^{\mathcal{U}}_{1,L} \in \mathbb{R}^{d_{\mathcal{X}}}$, and feed it to a randomly initialized softmax classifier (i.e., the task-specific output layer), where the class probabilities are computed as $\text{softmax}((\boldsymbol{h}^{\mathcal{U}}_{1,L})^{\mathrm{T}} \boldsymbol{\Theta}_{\mathcal{S}}) \in \mathbb{P}^{n_{\mathcal{C}}}$, where $\boldsymbol{\Theta}_{\mathcal{S}} \in \mathbb{R}^{d_{\mathcal{X}} \times n_{\mathcal{C}}}$ is a parameter matrix, and $n_{\mathcal{C}}$ is the number of categories.
  We maximize the likelihood of the labeled training data by updating the parameters of the pre-trained LM $\boldsymbol{\Theta}$ and the added softmax classifier $\boldsymbol{\Theta}_{\mathcal{S}}$.
- **NLG task.**
  - Pack the source sequence $\mathcal{U}_{\mathrm{S}}$ and target sequence $\mathcal{U}_{\mathrm{T}}$ together with special tokens [SOS] and [EOS], to form the input sequence
$$\mathcal{U} = < [\text{SOS}], \mathcal{U}_{\mathrm{S}}, [\text{EOS}], \mathcal{U}_{\mathrm{T}}, [\text{EOS}] >$$
  - Mask some percentage of tokens in the target segment $\mathcal{U}_{\mathrm{T}}$ at random, and learning to recover the masked words by maximizing the likelihood of masked tokens given context.
  - The second [EOS] must be masked during fine-tuning because it worths nothing, except marking the end of the target sequence.
  - **($*$ Open to question)**The first [EOS] (i.e., between $\mathcal{U}_{\mathrm{S}}$ and $\mathcal{U}_{\mathrm{T}}$) must not be masked during fine-tuning because it contains the context information of source sequence.

34.8.4 Cognition

Reading Materials.

- Unified Language Model Pre-training for Natural Language Understanding and Generation (Dong et al., 2019)[18].

34.9 Generalized Autoregressive Pretraining for Language Understanding (XLNet)

34.9.1 Motivation

- Relying on corrupting the input with masks, BERT (Devlin et al., 2018)[14] neglects dependency between the masked positions and suffers from a pretrain-finetune discrepancy.

34.9.2 Contribution

- Stands as a generalized autoregressive pre-training method that enabling learning bidirectional contexts by maximizing the expected likelihood over all permutations of the factorization order $\mathcal{I} \sim \mathcal{Z}_{n_\mathcal{U}}$.
- Overcomes the limitations of BERT thanks to its autoregressive formulation.
- Integrates ideas from Transformer-XL (Dai el al., 2019)[42] into pretraining.

34.9.3 Definition (Not Accomplished)

Autoregressive (AR) LM & Autoencoding (AE) LM. Unsupervised representation learning has been highly successful in the domain of NLP. Typically, these methods first pretrain neural networks on large-scale unlabeled text corpora, and then finetune the models or representations on downstream tasks. Under this shared high-level idea, autoregressive (AR) language modeling (e.g., GPT (Radford et al., 2018)[15]) and autoencoding (AE) language modelling (e.g., BERT (Devlin et al., 2018)[14]) have been the two most successful pretraining objectives.

The objective of ARLM shows as follows:

$$\begin{aligned}\max_{\Theta} \ \mathcal{J}(\Theta;\mathcal{U}) &= \log \Pr(\mathcal{U};\Theta) \\ &= \sum_{j=2}^{n_\mathcal{U}} \log \Pr\left(u_j |\, u_{j':1\leq j':\leq j-1};\Theta\right) \\ &= \sum_{j=2}^{n_\mathcal{U}} \log \frac{\exp\left((\boldsymbol{h}_j^\mathcal{U})^\mathrm{T}\boldsymbol{x}_j^\mathcal{U}\right)}{\sum_{c=1}^{n_\mathcal{V}} \exp\left((\boldsymbol{h}_j^\mathcal{U})^\mathrm{T}\boldsymbol{x}_c^\mathcal{V}\right)}\end{aligned}$$

where $\boldsymbol{h}_j^\mathcal{U}$ is a contextual representation produced by neural models, such as RNNs or Transformers, and $\boldsymbol{x}_j^\mathcal{U}$ and $\boldsymbol{x}_c^\mathcal{V}$ denotes the token embeddings.

In comparison, BERT is based on denoising AELM. Specifically, for a original token sequence $\mathcal{U} = (u_1, \cdots, u_{n_\mathcal{U}})$, BERT first constructs a corrupted version $\mathcal{U}_\mathcal{M}$ by randomly setting a portion (e.g., 15%) of tokens in sequence $\mathcal{U}$ to a special symbol [MASK]. Let's note the set of masked tokens be $\mathcal{M}$. For example:

- $\mathcal{U}$ = "New York is a city".
- $\mathcal{U}_\mathcal{M}$ = "New [Mask] is a [Mask]".
- $\mathcal{M} = \{\text{York}, \text{city}\}$.

The training objective is to reconstruct all $u \in \bar{\mathcal{U}}$ from $\widetilde{\mathcal{U}}$:

$$\begin{aligned}\max_{\Theta} \ \mathcal{J}(\Theta;\mathcal{U}) &= \log \Pr(\,\mathcal{M}\,|\,\mathcal{U}_\mathcal{M};\Theta) \\ &\approx \sum_{u\in\mathcal{M}} \log \Pr(u\,|\,\mathcal{U}_\mathcal{M};\Theta) \quad \text{(Based on conditional independence assumption of all masked tokens } u\in\mathcal{M}\text{)} \\ &= \sum_{j=1}^{n_\mathcal{U}} m_j \log \Pr(u_j|\,\mathcal{U}_\mathcal{M};\Theta) \quad (u_j \text{ comes from } \mathcal{U}, \text{ not } \mathcal{U}_\mathcal{M} \text{ or } \mathcal{M}) \\ &= \sum_{j=1}^{n_\mathcal{U}} m_j \log \frac{\exp\left((\boldsymbol{h}_j^{\mathcal{U}_\mathcal{M}})^\mathrm{T}\boldsymbol{x}_j^\mathcal{U}\right)}{\sum_{c=1}^{n_\mathcal{V}} \exp\left((\boldsymbol{h}_j^{\mathcal{U}_\mathcal{M}})^\mathrm{T}\boldsymbol{x}_c^\mathcal{V}\right)}\end{aligned}$$

where $m_j = 1$ indicates $u_j \in \mathcal{U}$ is masked. $\boldsymbol{h}_j^{\mathcal{U}_\mathcal{M}} \in \mathbb{R}^{d_\mathcal{X}}$ represents the encoding vector of $u_j \in \mathcal{U}$ with contextual information.

The pros and cons of the two pretraining objectives are compared in the following aspects:

- **Conditional Independence Assumption.**
  - AE language models (e.g., BERT) establishs the joint conditional probability $\Pr(\mathcal{M}\,|\,\mathcal{U}_{\mathcal{M}};\Theta) \approx \prod_{u\in\mathcal{M}} \Pr(u|\,\mathcal{U}_{\mathcal{M}};\Theta)$ based on an ***conditional independence assumption of all masked tokens*** that all masked tokens $u \in \mathcal{M}$ are separately reconstructed. e.g., a sequence "It shows that the housing crisis was turned into a banking crisis", if we mask "banking" and "crisis", when using MLM, we only use the tokens not be masked to predict "banking" and "crisis", therefore we lost the dependency between "banking" and "crisis".
  - AR language models (e.g., GPT) establishs th objective $\Pr(\mathcal{U};\Theta) = \prod_{j=2}^{n_{\mathcal{U}}} \Pr(u_j|\,u_{j':1\le j'\le j-1};\Theta)$ without any independence assumption.
- **Input Noise.**
  - AE language models (e.g., BERT) contains artificial symbols like [MASK] that never occur in downstream tasks in fine-tuning, which creates a ***pretrain-finetune discrepancy***.
  - AR language models (e.g., GPT) does not rely on any input corruption and does not suffer from this issue.
- **Context Dependency.**
  - AE language models (e.g., BERT) encodes $\boldsymbol{h}_j^{\mathcal{U}_{\mathcal{M}}}$ with contextual information on both sides, therefore its objective allows the model to be pretrained to better capture bidirectional context.
  - AR language models (e.g., GPT) encodes $\boldsymbol{h}_j^{\mathcal{U}}$ only conditioned on the tokens up to position j.

Objective: Permutation Language Modeling. Authors propose the permutation language modeling objective that not only retains the benefits of AR models but also allows models to capture bidirectional contexts like AE models.

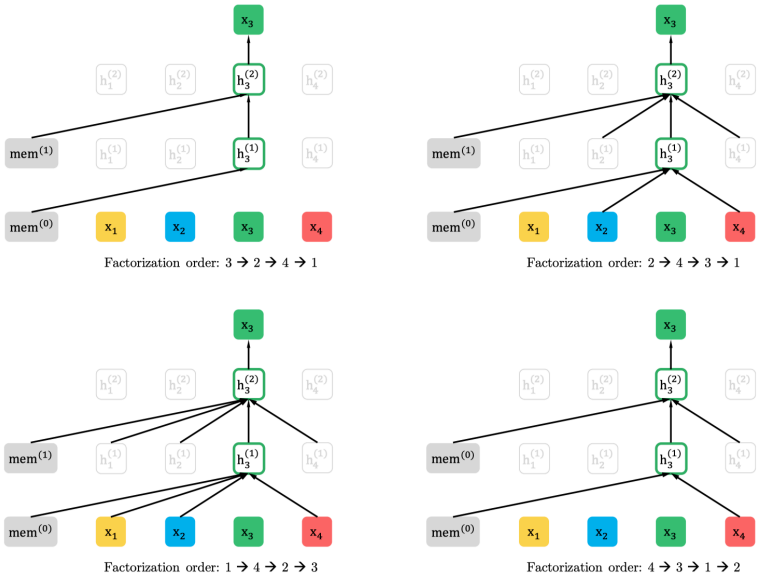


Figure 52: Illustration of the permutation language modeling objective for predicting u_3 ($\boldsymbol{x}_3$ as embedding) given the same input sequence $\mathcal{U}$ but with different factorization orders $\mathcal{I} \in \mathcal{Z}_{n_{\mathcal{U}}}$. ***In practise, we can get PLM by setting different mask matrix $\boldsymbol{M}_{\mathcal{I}}$***. Source: (Yang et al., 2019)[19].

Let $\mathcal{Z}_{n_{\mathcal{U}}}$ be the set of all possible permutations of the $n_{\mathcal{U}}$-length index sequence $(1, 2, \cdots, n_{\mathcal{U}})$, $\mathcal{I}$ is a $n_{\mathcal{U}}$-length index sequence sampled from the set $\mathcal{Z}_{n_{\mathcal{U}}}$ and $\mathcal{I}_j$ means the j-th element (i.e., a index) in sequence $\mathcal{I}$, $\mathcal{I}_{<j}$ means the first $j-1$ elements in sequence $\mathcal{I}$. For example:

- $\mathcal{U}$ = "New York is modern".
- $n_{\mathcal{U}} = 4$.
- $\mathcal{Z}_{n_{\mathcal{U}}} = \mathcal{Z}_4 = \underbrace{\{(1,2,3,4),(1,2,4,3),(1,3,2,4),(1,3,4,2),\cdots,(4,3,1,2),(4,3,2,1)\}}_{n_{\mathcal{U}}!\text{ index sequences}}$.
- $\mathcal{I} = (3,1,2,4) \sim \mathcal{Z}_{n_{\mathcal{U}}}$ under the uniform distribution.

- $\mathcal{I}_3 = 2$ and $\mathcal{I}_{<3} = (3, 1)$.
- $u_{\mathcal{I}_3} = u_2 =$ York and $u_{j':j' \in \mathcal{I}_{<3}} = u_{j':j' \in (3,1)} =$ "is New"

Then the proposed permutation language modeling objective can be expressed as follows:

$$\max_{\Theta} \ \mathcal{J}(\Theta; \mathcal{U}) = \mathbb{E}_{\mathcal{I} \sim \mathcal{Z}_{n_\mathcal{U}}} \left[\sum_{j=2}^{n_\mathcal{U}} \log \Pr \left(u_{\mathcal{I}_j} | u_{j':j' \in \mathcal{I}_{<j}}; \Theta \right) \right]$$

Essentially, for a token sequence $\mathcal{U}$, we sample a factorization order $\mathcal{I} \sim \mathcal{Z}_{n_\mathcal{U}}$ at a time and decompose the likelihood $\Pr(\mathcal{U}; \Theta)$ according to factorization order $\mathcal{I}$. Since the same model parameter Θ is shared across all factorization orders $\mathcal{I} \in \mathcal{Z}_{n_\mathcal{U}}$ during training, in expectation, u_j has seen every possible element $u_{j'} \neq u_j$ in the sequence, hence being able to ***capture the bidirectional contextual information***. Moreover, as this objective fits into the AR framework, it naturally ***avoids the conditional independence assumption of masked tokens and the pretrain-finetune discrepancy***.

Note after permutation for sequence $\mathcal{U}$, we still use the ***same positional encodings corresponding to the original sequence*** $\mathcal{U}$.

Architecture: Two-Stream Self-Attention for Target-Aware Representations. While the permutation language modeling objective has desired properties, naive implementation with standard Transformer parameterization may not work.

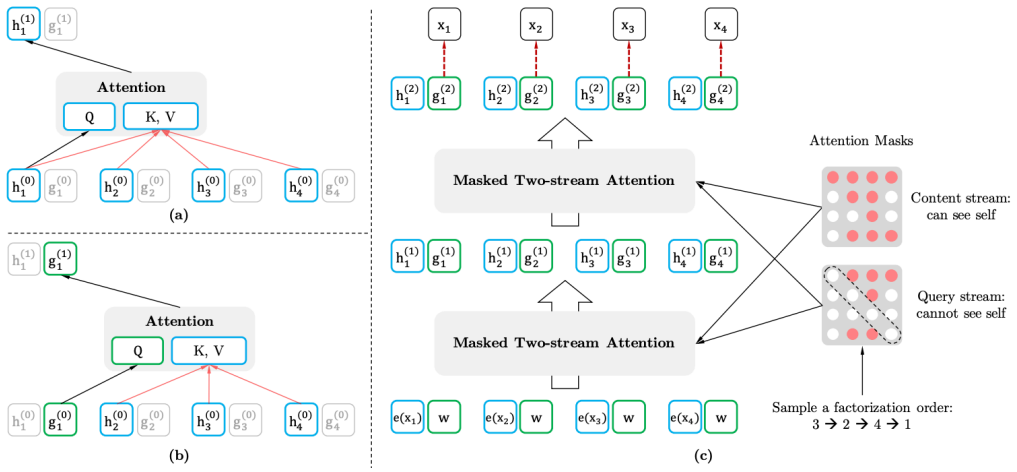


Figure 53: (a): Content stream attention, which is the same as the standard self-attention. (b): Query stream attention, which does not have access information about the content $u_{\mathcal{I}_j}$. (c): Overview of the permutation language modeling training with two-stream attention. Source: (Yang et al., 2019)[19].

Consider two different permutations $\mathcal{I}^{(1)}, \mathcal{I}^{(2)} \sim \mathcal{Z}_{n_\mathcal{U}}$ satisfying the following relationship:

$$\mathcal{I}^{(1)}_{<j} = \mathcal{I}^{(2)}_{<j} \quad \text{but} \quad \mathcal{I}^{(1)}_{j} \neq \mathcal{I}^{(2)}_{j}.$$

substituting the two permutations respectively into the naive parameterization, we have:

$$\Pr()$$

34.9.4 Cognition

Reading Materials.

- XLNet: Generalized Autoregressive Pretraining for Language Understanding (Yang et al., 2019)[19].

34.10 Retrieval-Augmented Language Model Pre-Training (REALM)

34.10.1 Motivation

34.10.2 Contribution

34.10.3 Definition

34.10.4 Cognition

Reading Materials.

- REALM: Retrieval-Augmented Language Model Pre-Training (Guu et al., 2020)[64].

35 Semantic Matching

36 Neural Machine Translation

Appendices

Appendix A Linear Algebra

A.1 Matrices

A.1.1 Matrix Product Operations

Frobenius product Officially, using $\cdot$ to represent Frobenius product. Given two real number-valued $n \times m$ matrices $\boldsymbol{A}$ and $\boldsymbol{B}$, written explicitly as

$$\boldsymbol{A} = \begin{pmatrix} \mathrm{A}_{11} & \mathrm{A}_{12} & \cdots & \mathrm{A}_{1m} \\ \mathrm{A}_{21} & \mathrm{A}_{22} & \cdots & \mathrm{A}_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ \mathrm{A}_{n1} & \mathrm{A}_{n2} & \cdots & \mathrm{A}_{nm} \end{pmatrix}, \ \boldsymbol{B} = \begin{pmatrix} \mathrm{B}_{11} & \mathrm{B}_{12} & \cdots & \mathrm{B}_{1m} \\ \mathrm{B}_{21} & \mathrm{B}_{22} & \cdots & \mathrm{B}_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ \mathrm{B}_{n1} & \mathrm{B}_{n2} & \cdots & \mathrm{B}_{nm} \end{pmatrix}$$

the Frobenius inner product is defined by the following summation $\sum$ of matrix elements,

$$\begin{aligned} \boldsymbol{A} \cdot \boldsymbol{B} =< \boldsymbol{A}, \boldsymbol{B} >_{\mathrm{F}} &= \sum_{i=1}^{n} \sum_{j=1}^{m} \mathrm{A}_{ij} \mathrm{B}_{ij} \\ &= \mathrm{vec}(\boldsymbol{A})^{\mathrm{T}} \mathrm{vec}(\boldsymbol{B}) \\ &= \mathrm{tr}(\boldsymbol{A}^{\mathrm{T}} \boldsymbol{B}) \end{aligned}$$

Proof. Calculation of $\mathrm{tr}(\boldsymbol{A}^{\mathrm{T}} \boldsymbol{B})$ is shown as follows:

$$\begin{aligned} \because \boldsymbol{A}^{\mathrm{T}} \boldsymbol{B} &= \begin{pmatrix} \mathrm{A}_{11} & \mathrm{A}_{21} & \cdots & \mathrm{A}_{n1} \\ \mathrm{A}_{12} & \mathrm{A}_{22} & \cdots & \mathrm{A}_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ \mathrm{A}_{1m} & \mathrm{A}_{2m} & \cdots & \mathrm{A}_{nm} \end{pmatrix} \begin{pmatrix} \mathrm{B}_{11} & \mathrm{B}_{12} & \cdots & \mathrm{B}_{1m} \\ \mathrm{B}_{21} & \mathrm{B}_{22} & \cdots & \mathrm{B}_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ \mathrm{B}_{n1} & \mathrm{B}_{n2} & \cdots & \mathrm{B}_{nm} \end{pmatrix} \\ &= \begin{pmatrix} \sum_{i=1}^{n} \mathrm{A}_{i1} \mathrm{B}_{i1} & \sum_{i=1}^{n} \mathrm{A}_{i1} \mathrm{B}_{i2} & \cdots & \sum_{i=1}^{n} \mathrm{A}_{i1} \mathrm{B}_{im} \\ \sum_{i=1}^{n} \mathrm{A}_{i2} \mathrm{B}_{i1} & \sum_{i=1}^{n} \mathrm{A}_{i2} \mathrm{B}_{i2} & \cdots & \sum_{i=1}^{n} \mathrm{A}_{i2} \mathrm{B}_{im} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{i=1}^{n} \mathrm{A}_{im} \mathrm{B}_{i1} & \sum_{i=1}^{n} \mathrm{A}_{im} \mathrm{B}_{i2} & \cdots & \sum_{i=1}^{n} \mathrm{A}_{im} \mathrm{B}_{im} \end{pmatrix} \in \mathbb{R}^{m \times m} \\ \therefore \mathrm{tr}(\boldsymbol{A}^{\mathrm{T}} \boldsymbol{B}) &= \sum_{j=1}^{m} \sum_{i=1}^{n} \mathrm{A}_{ij} \mathrm{B}_{ij} \\ &= \sum_{i=1}^{n} \sum_{j=1}^{m} \mathrm{A}_{ij} \mathrm{B}_{ij} \\ &= \boldsymbol{A} \cdot \boldsymbol{B} \end{aligned}$$

Hadamard product Officially, using $\odot$ to represent Hadamard product. Given two real number-valued $n \times m$ matrices $\boldsymbol{A}$ and $\boldsymbol{B}$, written explicitly as

$$\boldsymbol{A} = \begin{pmatrix} \mathrm{A}_{11} & \mathrm{A}_{12} & \cdots & \mathrm{A}_{1m} \\ \mathrm{A}_{21} & \mathrm{A}_{22} & \cdots & \mathrm{A}_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ \mathrm{A}_{n1} & \mathrm{A}_{n2} & \cdots & \mathrm{A}_{nm} \end{pmatrix}, \ \boldsymbol{B} = \begin{pmatrix} \mathrm{B}_{11} & \mathrm{B}_{12} & \cdots & \mathrm{B}_{1m} \\ \mathrm{B}_{21} & \mathrm{B}_{22} & \cdots & \mathrm{B}_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ \mathrm{B}_{n1} & \mathrm{B}_{n2} & \cdots & \mathrm{B}_{nm} \end{pmatrix}$$

the Hadamard product is defined by the following element-wise product:

$$\boldsymbol{A} \odot \boldsymbol{B} = \begin{pmatrix} \mathrm{A}_{11} \mathrm{B}_{11} & \mathrm{A}_{12} \mathrm{B}_{12} & \cdots & \mathrm{A}_{1m} \mathrm{B}_{1m} \\ \mathrm{A}_{21} \mathrm{B}_{21} & \mathrm{A}_{22} \mathrm{B}_{22} & \cdots & \mathrm{A}_{2m} \mathrm{B}_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ \mathrm{A}_{n1} \mathrm{B}_{n1} & \mathrm{A}_{n2} \mathrm{B}_{n2} & \cdots & \mathrm{A}_{nm} \mathrm{B}_{nm} \end{pmatrix}$$

Kronecker product

Appendix B Tensor Calculus

B.1 Differentiation of Univariate Functions

B.2 Partial Differentiation and Gradients

Consider a general case where the function $f(\cdot)$ depends on one or more variables $\boldsymbol{x} \in \mathbb{R}^n$, e.g., $f(\boldsymbol{x}) = f(\mathrm{x}_1, \mathrm{x}_2)$. The generalization of the derivative to functions of several variables not univariable is the gradient (Deisenroth et al., 2019)[65].

Definition B.1 (Partial Derivative)**.** *For a function $f : \mathbb{R}^n \to \mathbb{R}, \boldsymbol{x} \mapsto f(\boldsymbol{x}), \boldsymbol{x} \in \mathbb{R}^n$ of n variables $\mathrm{x}_1, \ldots, \mathrm{x}_n$ we define the partial derivatives as:*

$$\begin{aligned}\frac{\partial f(\boldsymbol{x})}{\partial \mathrm{x}_1} &= \lim_{h \to 0} \frac{f(\mathrm{x}_1 + h, \mathrm{x}_2, \cdots, \mathrm{x}_n) - f(\boldsymbol{x})}{h} \\ &\vdots \\ \frac{\partial f(\boldsymbol{x})}{\partial \mathrm{x}_n} &= \lim_{h \to 0} \frac{f(\mathrm{x}_1, \mathrm{x}_2, \cdots, \mathrm{x}_n + h) - f(\boldsymbol{x})}{h}\end{aligned}$$

Definition B.2 (Gradient)**.** *Collect all partial derivative of $f(\boldsymbol{x})$ with respect to $\mathrm{x}_1, \ldots, \mathrm{x}_n$ in a row vector:*

$$\nabla_{\boldsymbol{x}} f = \operatorname{grad} f = \frac{\mathrm{d}f}{\mathrm{d}\boldsymbol{x}} = \left[\frac{\partial f(\boldsymbol{x})}{\partial \mathrm{x}_1}, \frac{\partial f(\boldsymbol{x})}{\partial \mathrm{x}_2}, \cdots, \frac{\partial f(\boldsymbol{x})}{\partial \mathrm{x}_n}\right] \in \mathbb{R}^{1 \times n}$$

where n is called the number of variables and 1 is the dimension of the image range/codomain of f. Here we define the column vector $\boldsymbol{x} = [\mathrm{x}_1, \mathrm{x}_2, \cdots, \mathrm{x}_n]^{\mathrm{T}} \in \mathbb{R}^n$. Above row vector $\nabla_{\boldsymbol{x}} f$ is called the gradient of f or the Jacobian and is the generalization of the derivative from Subsection B.1.

Appendix C Probability and Distributions

C.1 Discrete Probability Distributions

C.1.1 Total Variation Distance

C.1.2 Hellinger Distance

C.1.3 χ^2 and Kullback-Leibler Divergences

Reading Materials.

- A short note on learning discrete distributions (Canonne, 2020.02).

Appendix D Stochastic Processes

D.1 Markov Decision Processes (MDPs)

D.1.1 Definition of MDP

In reinforcement learning, the interactions between the agent and the environment are often described by a Markov Decision Process (MDP)[66], specied by a 6-tuple: $\mathcal{M} =< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma, \mu >$, where:

- $\mathcal{S}$: A finite set of states. $s \in \mathcal{S}$.
- $\mathcal{A}$: A finite set of actions. $a \in \mathcal{A}$
- $\mathcal{P} : \mathcal{S} \times \mathcal{A} \to \mathbb{P}^{|\mathcal{S}|}$: Transition dynamics. $\mathcal{P}(s_{t+1}|s_t, a_t)$ maps a state-action pair at time t onto a probability distribution over states at time $t + 1$.
- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$: Reward function. $\mathcal{R}_t = \mathcal{R}(s_t, a_t, s_{t+1})$ is the immediate reward associated with taking action a_t in state s_t at timestep t, and transitioning into state s_{t+1}.
- $\gamma \in [0, 1]$: Discount factor, which defines a horizon for the problem.
- $\mu \in \mathbb{P}^{|\mathcal{S}|}$: Initial probability distribution over states.

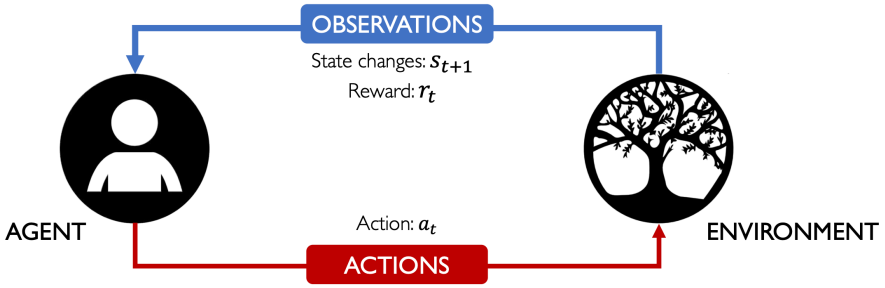


Figure 54: The perception-action-learning loop. At time t, the agent receives state s_t from the environment. The agent uses its policy to choose an action a_t. Once the action is executed, the environment transitions a step, providing the next state s_{t+1} as well as feedback in the form of a reward $\mathcal{R}_t$. The agent uses knowledge of state transitions, of the form $(s_t, a_t, s_{t+1}, \mathcal{R}_t)$, in order to learn and improve its policy.

D.1.2 Policy & Objective

D.1.3 Bellman Equations

D.2 Partially Observable Markov Decision Processes (POMDPs)

Appendix E Mathematical Optimization

Appendix F Algorothm Design & Analysis

Appendix G Programming Technology

G.1 C++

G.1.1 Streaming SIMD Extensions (SSE)

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 265–283. USENIX Association, 2016.

[2] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[3] François Chollet et al. Keras. https://keras.io, 2015.

[4] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.

[5] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermüller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, Yoshua Bengio, Arnaud Bergeron, James Bergstra, Valentin Bisson, Josh Bleecher Snyder, Nicolas Bouchard, Nicolas Boulanger-Lewandowski, Xavier Bouthillier, Alexandre de Brébisson, Olivier Breuleux, Pierre Luc Carrier, Kyunghyun Cho, Jan Chorowski, Paul F. Christiano, Tim Cooijmans, Marc-Alexandre Côté, Myriam Côté, Aaron C. Courville, Yann N. Dauphin, Olivier Delalleau, Julien Demouth, Guillaume Desjardins, Sander Dieleman, Laurent Dinh, Melanie Ducoffe, Vincent Dumoulin, Samira Ebrahimi Kahou, Dumitru Erhan, Ziye Fan, Orhan Firat, Mathieu Germain, Xavier Glorot, Ian J. Goodfellow, Matthew Graham, Çaglar Gülçehre, Philippe Hamel, Iban Harlouchet, Jean-Philippe Heng, Balázs Hidasi, Sina Honari, Arjun Jain, Sébastien Jean, Kai Jia, Mikhail Korobov, Vivek Kulkarni, Alex Lamb, Pascal Lamblin, Eric Larsen, César Laurent, Sean Lee, Simon Lefrançois, Simon Lemieux, Nicholas Léonard, Zhouhan Lin, Jesse A. Livezey, Cory Lorenz, Jeremiah Lowin, Qianli Ma, Pierre-Antoine Manzagol, Olivier Mastropietro, Robert McGibbon, Roland Memisevic, Bart van Merriënboer, Vincent Michalski, Mehdi Mirza, Alberto Orlandi, Christopher Joseph Pal, Razvan Pascanu, Mohammad Pezeshki, Colin Raffel, Daniel Renshaw, Matthew Rocklin, Adriana Romero, Markus Roth, Peter Sadowski, John Salvatier, François Savard, Jan Schlüter, John Schulman, Gabriel Schwartz, Iulian Vlad Serban, Dmitriy Serdyuk, Samira Shabanian, Étienne Simon, Sigurd Spieckermann, S. Ramana Subramanyam, Jakub Sygnowski, Jérémie Tanguay, Gijs van Tulder, Joseph P. Turian, Sebastian Urban, Pascal Vincent, Francesco Visin, Harm de Vries, David Warde-Farley, Dustin J. Webb, Matthew Willson, Kelvin Xu, Lijun Xue, Li Yao, Saizheng Zhang, and Ying Zhang. Theano: A python framework for fast computation of mathematical expressions. *CoRR*, abs/1605.02688, 2016.

[6] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830, 2011.

[7] LLC Gurobi Optimization. Gurobi optimizer reference manual, 2019.

[8] MOSEK ApS. *The MOSEK optimization toolbox for Python manual. Version 9.0.*, 2019.

[9] Philippe Laborie, Jérôme Rogerie, Paul Shaw, and Petr Vilím. Ibm ilog cp optimizer for scheduling. *Constraints*, 23(2):210–250, April 2018.

[10] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, pages 815–823. IEEE Computer Society, 2015.

[11] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[12] Yuxin Wu and Kaiming He. Group normalization. In Ferrari et al. [67], pages 3–19.

[13] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A. Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017*, pages 2242–2251. IEEE Computer Society, 2017.

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[15] Alec Radford. Improving language understanding by generative pre-training. *OpenAI*, 2018.

[16] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI*, 2018.

[17] Mandar Joshi, Danqi Chen, Yinhan Liu, Daniel S. Weld, Luke Zettlemoyer, and Omer Levy. Spanbert: Improving pre-training by representing and predicting spans. *CoRR*, abs/1907.10529, 2019.

[18] Li Dong, Nan Yang, Wenhui Wang, Furu Wei, Xiaodong Liu, Yu Wang, Jianfeng Gao, Ming Zhou, and Hsiao-Wuen Hon. Unified language model pre-training for natural language understanding and generation. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, pages 13042–13054, 2019.

[19] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime G. Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. Xlnet: Generalized autoregressive pretraining for language understanding. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, pages 5754–5764, 2019.

[20] Tsung-Yi Lin, Priya Goyal, Ross B. Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. In *IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017*, pages 2999–3007. IEEE Computer Society, 2017.

[21] Elad Hoffer and Nir Ailon. Deep metric learning using triplet network. In Bengio and LeCun [68].

[22] Weifeng Ge, Weilin Huang, Dengke Dong, and Matthew R. Scott. Deep metric learning with hierarchical triplet loss. In Ferrari et al. [67], pages 272–288.

[23] H. Robbins and S. Monro. A stochastic approximation method. *Annals of Mathematical Statistics*, 22:400–407, 1951.

[24] Sebastian Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016.

[25] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, pages III–1139–III–1147. JMLR.org, 2013.

[26] John C. Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12:2121–2159, 2011.

[27] Yann N. Dauphin, Harm de Vries, Junyoung Chung, and Yoshua Bengio. Rmsprop and equilibrated adaptive learning rates for non-convex optimization. *CoRR*, abs/1502.04390, 2015.

[28] Matthew D. Zeiler. ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701, 2012.

[29] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Bengio and LeCun [68].

[30] Jianbang Ding, Xuancheng Ren, Ruixuan Luo, and Xu Sun. An adaptive and momental bound method for stochastic learning. *CoRR*, abs/1910.12249, 2019.

[31] James Martens and Roger B. Grosse. Optimizing neural networks with kronecker-factored approximate curvature. In Francis R. Bach and David M. Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 2408–2417. JMLR.org, 2015.

[32] Vineet Gupta, Tomer Koren, and Yoram Singer. Shampoo: Preconditioned stochastic tensor optimization. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 1837–1845. PMLR, 2018.

[33] Rohan Anil, Vineet Gupta, Tomer Koren, Kevin Regan, and Yoram Singer. Second order optimization made practical. *CoRR*, abs/2002.09018, 2020.

[34] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[35] Arvind Neelakantan, Luke Vilnis, Quoc V. Le, Ilya Sutskever, Lukasz Kaiser, Karol Kurach, and James Martens. Adding gradient noise improves learning for very deep networks. *CoRR*, abs/1511.06807, 2015.

[36] Nitish Shirish Keskar and Richard Socher. Improving generalization performance by switching from adam to SGD. *CoRR*, abs/1712.07628, 2017.

[37] Zhilin Yang, Thang Luong, Russ R Salakhutdinov, and Quoc V Le. Mixtape: Breaking the softmax bottleneck efficiently. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 15922–15930. Curran Associates, Inc., 2019.

[38] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29:1189–1232, 2000.

[39] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi, editors, *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pages 785–794. ACM, 2016.

[40] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778. IEEE Computer Society, 2016.

[41] Lei Jimmy Ba, Ryan Kiros, and Geoffrey E. Hinton. Layer normalization. *CoRR*, abs/1607.06450, 2016.

[42] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G. Carbonell, Quoc Viet Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. In Anna Korhonen, David R. Traum, and Lluís Màrquez, editors, *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 2978–2988. Association for Computational Linguistics, 2019.

[43] Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. In *International Conference on Learning Representations*, 2020.

[44] Aidan N. Gomez, Mengye Ren, Raquel Urtasun, and Roger B. Grosse. The reversible residual network: Backpropagation without storing activations. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, pages 2214–2224, 2017.

[45] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Francis R. Bach and David M. Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 448–456. JMLR.org, 2015.

[46] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[47] Steffen Rendle. Factorization machines. In Geoffrey I. Webb, Bing Liu, Chengqi Zhang, Dimitrios Gunopulos, and Xindong Wu, editors, *ICDM 2010, The 10th IEEE International Conference on Data Mining, Sydney, Australia, 14-17 December 2010*, pages 995–1000. IEEE Computer Society, 2010.

[48] Steffen Rendle. Factorization machines with libfm. *ACM TIST*, 3(3):57:1–57:22, 2012.

[49] Yu-Chin Juan, Yong Zhuang, Wei-Sheng Chin, and Chih-Jen Lin. Field-aware factorization machines for CTR prediction. In Shilad Sen, Werner Geyer, Jill Freyne, and Pablo Castells, editors, *Proceedings of the 10th ACM Conference on Recommender Systems, Boston, MA, USA, September 15-19, 2016*, pages 43–50. ACM, 2016.

[50] Yuchin Juan, Damien Lefortier, and Olivier Chapelle. Field-aware factorization machines in a real-world online advertising system. In Rick Barrett, Rick Cummings, Eugene Agichtein, and Evgeniy Gabrilovich, editors, *Proceedings of the 26th International Conference on World Wide Web Companion, Perth, Australia, April 3-7, 2017*, pages 680–688. ACM, 2017.

[51] Bowen Yuan, Meng-Yuan Yang, Jui-Yang Hsia, Hong Zhu, Zhirong Liu, Zehnhua Dong, and Chih-Jen Lin. One-class field-aware factorization machines for recommender systems with implicit feedbacks. *None*, 09 2019.

[52] Weiping Song, Chence Shi, Zhiping Xiao, Zhijian Duan, Yewen Xu, Ming Zhang, and Jian Tang. Autoint: Automatic feature interaction learning via self-attentive neural networks. In Wenwu Zhu, Dacheng Tao, Xueqi Cheng, Peng Cui, Elke A. Rundensteiner, David Carmel, Qi He, and Jeffrey Xu Yu, editors, *Proceedings of the 28th ACM International Conference on Information and Knowledge Management, CIKM 2019, Beijing, China, November 3-7, 2019*, pages 1161–1170. ACM, 2019.

[53] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 779–788. IEEE Computer Society, 2016.

[54] Joseph Redmon and Ali Farhadi. YOLO9000: better, faster, stronger. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 6517–6525. IEEE Computer Society, 2017.

[55] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *CoRR*, abs/1804.02767, 2018.

[56] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: single shot multibox detector. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part I*, volume 9905 of *Lecture Notes in Computer Science*, pages 21–37. Springer, 2016.

[57] Yunjey Choi, Min-Je Choi, Munyoung Kim, Jung-Woo Ha, Sunghun Kim, and Jaegul Choo. Stargan: Unified generative adversarial networks for multi-domain image-to-image translation. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 8789–8797. IEEE Computer Society, 2018.

[58] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In Christopher J. C. Burges, Léon Bottou, Zoubin Ghahramani, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*, pages 3111–3119, 2013.

[59] Xin Rong. word2vec parameter learning explained. *CoRR*, abs/1411.2738, 2014.

[60] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In Alessandro Moschitti, Bo Pang, and Walter Daelemans, editors, *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1532–1543. ACL, 2014.

[61] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. In Marilyn A. Walker, Heng Ji, and Amanda Stent, editors, *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2018, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 1 (Long Papers)*, pages 2227–2237. Association for Computational Linguistics, 2018.

[62] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019.

[63] Dan Hendrycks and Kevin Gimpel. Bridging nonlinearities and stochastic regularizers with gaussian error linear units. *CoRR*, abs/1606.08415, 2016.

[64] Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Ming-Wei Chang. Realm: Retrieval-augmented language model pre-training. *CoRR*, abs/2002.08909, 2020.

[65] Marc Peter Deisenroth, A. Aldo Faisal, and Cheng Soon Ong. *Mathematics for Machine Learning*. Cambridge University Press, 2020.

[66] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley Series in Probability and Statistics. Wiley, 1994.

[67] Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, and Yair Weiss, editors. *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part XIII*, volume 11217 of *Lecture Notes in Computer Science*. Springer, 2018.

[68] Yoshua Bengio and Yann LeCun, editors. *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.