Optimization

Gradient Descent

$\nabla f(w_0) = \frac{\partial f}{\partial w_0}$
$w_1 = w_0 - \eta \nabla f(w_0)$

Stochastic Gradient Descent

Randomly choose a datapoint, compute the gradient of it, move in the direction where the gradient is minimized in each iteration.

Mini-Batch Gradient Descent

Compute gradient on batches of 10-1000 samples.

Batch Gradient Descent

Compute gradient on the whole dataset.

Metrics

Accuracy: $\frac{TP+TN}{N}$
Error Rate: $\frac{FP+FN}{N}$
Recall: $\frac{TP}{\text{Actual Yes}}$
Precision: $\frac{TP}{Positive}$
Specificity: $\frac{TN}{\text{Actual No}}$
ROC Space: y-axis – TP Rate, x-axis, FP Rate - (0, 0) never issue a positive classification - (1, 1) always issue a positive classification - (0, 1) perfect classification - (1, 0) perfect misclassification The ROC curve tells us, given the same threshold, models perform better on the population distinct between classes.
AUC: If we pick a random positive and a random negative, what's the probability my model scores them in the correct relative order?

Feature Engineering

Feature engineering: a process of using domain knowledge to create new features from the raw data that might help a machine learning model perform better
Feature learning: a process used in machine learning where algorithms automatically identify and extract useful features or patterns from raw data

Feature Scaling

Min-max scaling: $\frac{x-min(X)}{range(X)}$, $x' \in [-1, 1]$ or $[0, 1]$
Standardization: $\frac{x-\bar{X}}{\sigma_X}$

The Binning Trick

Create several boolean bins, each mapping to a new unique feature. Allows model to fit a different value for each bin. Solve the limitations of linear model.

Non-linearly separable Features

If we cannot find a linear function to separate the feature, we can define a new feature $X_3 = X_1 \times X_2$ or $X_3 = X_1^2$ and fit into a linear function $y = f(b + w_1X_1 + w_2X_2 + w_3X_3)$

NLP Basics

Tokenization: split a document into a list of meaningful units
Stopwords: mostly connector words that do not contribute much to the meaning of the sentence
Lemmatization: turn the word into its root word
Stemming: remove inflectional forms of a word and bring them to a base form

N-gram model

Pros: word order is considered
Cons: vocab size is very huge, able to incorporate limited word order information

TFIDF

$TF(w) =$ Number of times the word w occurs in a document/Total number of words in the document
$IDF(w) = log$Total number of documents/Number of documents containing word w)
$weight(w, d) = TF(w, d) \times IDF(w)$

Cosine Similarity

$similarity(v, w) = cos\theta = \frac{v \cdot w}{\|v\|\|w\|}$

Regularization

Bias Variance Tradeoff

Bias: $Bias[\hat{f}(x)] = E_D[\hat{f}(x)] - f(x)$
Variance: $Var[\hat{f}(x)] = E_D[(\hat{f}(x) - E_D[\hat{f}(x)])^2]$

L2 Regularization

$Loss = Error(y, \hat{y}) + \lambda\|w\|_2^2$

L1 Regularization

$Loss = Error(y, \hat{y}) + \lambda\|w\|_1^2$

Classification

Logistic Regression

$\hat{y} = \frac{1}{1+e^{-(w^Tx+b)}}$

Cross-Entropy Loss

The lower entropy the better
$LogLoss(w) = -\sum_{n=1}^{N}\sum_{k=1}^{K} y_{ik}log(\hat{y}_{ik})$
Low probability event $\Rightarrow$ High Information (surprising) $\Rightarrow$ High entropy
High probability event $\Rightarrow$ Low Information (not surprising) $\Rightarrow$ Low entropy

Multinomial Logit

$\hat{y}_1 = \frac{e^{z_1}}{\sum_{k=1}^{K} e^{z_k}}$
$z_k = b_k + w_k^T x$

Naive Bayes

$H_{MAP} = argmax_{H\in\mathbb{H}}P(H|E) = argmax_{H\in\mathbb{H}}P(H)P(E|H)$
MAP is to find the class that bears the highest probability given features E.
If attributes are coditionally independence, $p(d_1|d_2, \ldots, d_n, h_i) = \Pi_{j=1}^{n} p(d_j|h_i)$
If $p(d_1, \ldots, d_n|h_i) \neq p(d_1|h_i) \times \ldots p(d_n|h_i)$, prediction is still equivalent as long as
$argmax_{h_i \in mathbbH} p(d_1, \ldots, d_n|h_i)p(h_i) = argmax_{h_i \in mathbbH} p(d_1|h_i) \ldots p(d_n|h_i)p(h_i)$

Support Vector Machine

MMH: MMH is the separating hyperplane for which the margin is largest
MMC: MMC is the classification based on MMH
Hyperplane equation: $\beta_0 + \beta_1X_1 + \ldots + \beta_pX_p = 0$
$D(x_1, x_2, L) = \frac{|f(x^*)|}{\sqrt{\|\beta\|}}$
The larger is $|f(x^*)|$, the further s^* from the hyperplane L.
$\beta_0 + \beta_1X_1 + \ldots + \beta_pX_p = 0 \Rightarrow x$ lies on the hyperplane
$\beta_0 + \beta_1X_1 + \ldots + \beta_pX_p < 0 \Rightarrow x$ lies on one side
$\beta_0 + \beta_1X_1 + \ldots + \beta_pX_p > 0 \Rightarrow x$ lies on another side
the magnitude of $|f(x^*)|$ is a measure of the confidence that the observation is assigned with the corresponding class
Hinge Loss: $\frac{1}{n}\sum_{i=1}^{n} max(0, 1 - y_i\frac{\beta^TX_i}{M})$

Decision Tree

Gini index: measures impurity of classes within the samples it applies to $G_i = 1 - \sum_{k=1}^{n} p_k^2$
How to find the best attribte A and its thresholds $r(A)$ to split? CART: A is the one that minimizes the Gini index by splitting set S on A. CART searches for the pair $(A, r(A))$ that produces the purest subsets.
$p(t)$ is the proportion of the number of elemetns in t to the number of elements in set S
$p_t(k)$ is the ratio of class k elements in subset t
$G(t) = 1 - \sum_{k=1}^{n} p_t^2(k)$
$G(S, A) = \sum_{t\in T} p(t)G(t)$
ID3: apply splitting only on unused attributes. Stop once it cannot find a split that will reduce entropy or the tree reaches the maximum depth or no more attributes or samples are available
Entropy $H(S) = -\sum_{k=1}^{n} p_s(k)log_2p_s(k)$ is a measure of the amount of information about the target in the set S
Information Gain:
$IG(S, A) = H(S) - \sum_{t\in T} p(t)H(t) = H(S) - H(S|A)$
Information gain is the expected reduction in entropy of target variable Y for data sample S, due to sorting on variable A
How to avoid overfitting? Cost Complexity Measure: $\alpha_{eff} = \frac{R(t)-R(T_t)}{|T|-1}$, this is the impurity after split vs before split, the smaller this value is, the useless the split is
Grow a full tree, and then perform post-pruning with minimal cost complexity pruning: A non-terminal node with the smallest value of α_{eff} is the weakest link and will be pruned.

Ensemble Learning

Bagging

Limitation: No distinction between "easy" examples and "difficult" examples. A constant weight for each classifier. No distinction between accurate classifiers and inaccurate classifiers.

Boosting

Adaboost

Instances that are wrongly classified will have their weights increased
Instances that are correctly classified will have their weights decreased
Misclassification rate:
$\epsilon_t = (\sum_{i=1}^{m} D_i, t)_{h_t(x_i)\neq y_i} / \sum_{i=1}^{m} D_{i,t}$
Weight for classifier: $\alpha_t = \frac{1}{2}ln\frac{1-\epsilon_t}{\epsilon_t}$
$D_{t+1}(i) = D_t(i)e^{\alpha_t}$

Gradient Boosting

$F = \sum_t \rho_t h_t$
$F = F + \rho h$, ρ is the learning rate
'shortcomings' (error) are identified by negative gradients.

Stacking

1.split the data into 2 sets training and test set
2.split the training data into K-folds just like K-fold CV
3.a base model is then fitted on the K-1 parts and predictions are mode for Kth part. Repeat for K times
4. the base model is then fitted on the whole train data set to calculate its performance on the test set
5. steps 3-4 are repeated for another base mdoel. Then, we are going to have another set of predictions for the train set and test set
6. the predictions from the train set are used as features to build a second-level model
7. this model is used to make final predictions on the test prediction set

DNN

- sigmoid: $f(z) = \frac{1}{1+e^{-z}}$
- ReLU: $f(z) = max(z, 0)$
- Tanh: $f(z) = \frac{2}{1+e^{-2z}} - 1$
- Leaky ReLU: $f(z) = 1(x < 0)(ax) + 1(x > 0)(x)$

NN with at least one hidden layer can approximate any continuous function.

One-vs-All Multi-Class

For each of K classes, compute their logit outputs ($\frac{1}{1+e^{-a_k}}$). End up with K predicted probability. Choose the highest one out of these K probabilities. They sum up exceeds 1.

Softmax Multi-Class

For each of K classes, they use the same softmax function ($\frac{exp(a_k)}{\sum_{k=1}^{K} exp(a_k)}$). In total, they sum up to 1.

Solutions of Overfitting NN

- Early stopping: stop updating the weights once the validation error starts increasing
- Regularization: $J = \frac{1}{N}\sum_{i=1}^{N} l(f_\theta(x_i), y_i) + \lambda g(\theta)$
- Dropout: randomly dropping out units in a network for a single gradient step. They come back with unchanged weights at the next step.
- Batch normalization: standardize the distribution of layer inputs and control the amount by which the hidden units shift

Word Embeddings

- Localist representation is one-hot encoding, it's a sparse matrix, no contextual/semantic information

Distributed Similarity

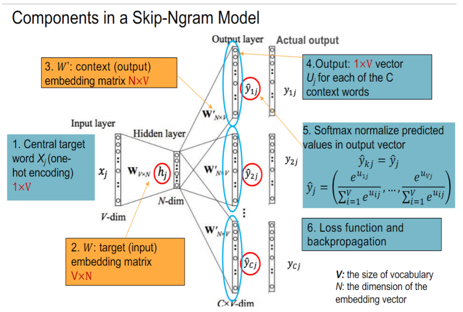
- words are represented by their context
- distributional similarities sue the set of context sin which words apperar to measure heir similarity

Word2Vec

- An NN that predict the next word
- Need to define target and context words (by using window size)
- Loss function: $H(\hat{y}_j, y_j) = -\sum_{i\in C} log\ p(x_{ij}|x_j) = -\sum_{i\in C} log\frac{e^{u_{j,i}}}{\sum_{n=1}^{V} e^{u_{j,n}}} = C \cdot log\sum_{n=1}^{V} e^{u_{j,n}} - \sum_{i\in C} u_{j,i}$
- Similar context will be assigned similar vectors

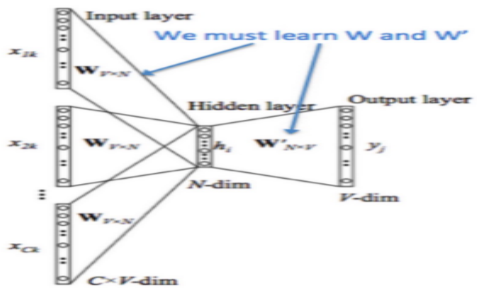
Skip-Ngram

- use central word to predict context word
- works well with small amount of the training data, represents well even rare words or phrases
1. generate one-hot encoding for the central target word
2. get word vector for the target X using the target embedding matrix
3. get c score vectors u via the dot product of word vector h_J and the context embedding matrix W'
4. turn score vectors to probabilities $y = softmax(u)$
5. compare with the true probabilities, which are the on-hot vectors of the actual output
6. update and repeat the above steps base don the loss


Components in a Skip-Ngram Model
Output layer
Actual output
3. W': context (output) embedding matrix N×V
4.Output: 1×V vector U_j for each of the C context words
5. Softmax normalize predicted values in output vector
6. Loss function and backpropagation
1. Central target word X_j (one-hot encoding) 1×V
2. W: target (input) embedding matrix V×N
Input layer
Hidden layer
V-dim
N-dim
C×V-dim
V: the size of vocabulary
N: the dimension of the embedding vector

CBoW

- learning to predict the word by the context
- several times faster to trian than the skip-gram, slightly better accuracy for the frequent words
1. generate one-hot encoding for the context words
2. get word vector h_J for each context owrd using the embedding matrix W. Then do average over these C vectors
3. get one score vector u using the embedding matrix W'
4. turn score vectors to probabilities $y = softmax(u)$
5. compare with the true probabilities, which are the one-hot vectors of the actual output (the particular central word)
6. update and repeat the above steps based on the loss



Negative Sampling

Negative sampling addresses this issue by randomly selecting a small number of "negative" examples (words not appearing in the context) for each target word during training. By doing this, instead of updating the model parameters for all words in the context, it updates parameters only for the target word and a small number of negative examples, significantly reducing the computational cost of training.

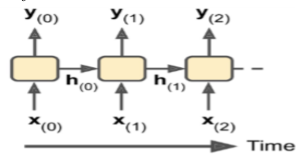
CNN

- Correlation: dot-multiply original matrix by a filter (kernel, correlation matrix)
$w(x, y) * f(x, y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s)\ f(x + s, y + t)$
- Convolution: rotate the filter by 180° then dot-multiply. Convolution is associative and commutative
$w(x, y) * f(x, y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s)\ f(x - s, y - t)$

- Paddings: add numbers (sometimes 0) to the edge of the original matrix, so that the feature map is the same size as before
- stride: step size of a sliding window
- Pooling: take mean (smoothing and reducing the impact of outliers) or max (retain the most prominent features in a region) of a given size matrix from the feature map

RNN

- Memory cell: preserves some state across time steps
- An RNN cell will preserve state h_{t-1}, take input x_t, then produce output y_t and update the preserved state to be h_t



- Output vector: $y_t = f(Vh_t + b_y)$
- Update hidden state: $h_t = g(Wx_t + Uh_{t-1} + b_h)$
- Input vector: x_t
- RNN feed each word at a time, while FNN feed all words concurrently
- RNN take the output from each time steps to compute loss, hence, there are T losses, total loss will be aggregrate over these T losses.
- Teacher forcing: the model is given the correct history sequence to predict the next word, rather than feeding the model its prediction from the previous time step
- Self-supervision: the training data is a corpus of text and at each time step t the model predict the next word

Stacked RNN

- stack multiple RNN cells to produce output for each time step

Problems of RNN

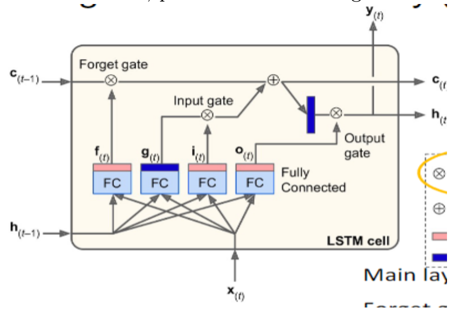
- When many values ¿ 1, exploding gradients
- When many values ¡ 1, vanishing gradients
  - can be solved by using ReLU activation function
  - or gradient clipping, constrain the value of gradient
- Long-term dependency problem

LSTM

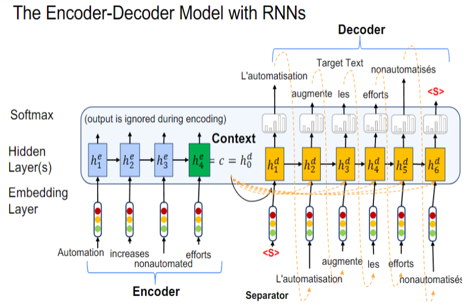
- Main layer: $g_t = tanh(W_{xg}^T \cdot x_t + W_{hg}^T \cdot h_{t-1} + b_g)$
- Forget gate: $f_t = \sigma(W_{xf}^T \cdot x_t + W_{hf}^T \cdot h_{t-1} + b_f)$
- Input gate: $i_t = \sigma(W_{xi}^T \cdot x_t + W_{hi}^T \cdot h_{t-1} + b_i)$
- Output gate: $o_t = \sigma(W_{xo}^T \cdot x_t + W_{ho}^T \cdot h_{t-1} + b_o)$
- Memory cell/Long-term state:
$c_t = f_t \otimes c_{t-1} + i_t \otimes g_t$

- first term: decide how many memory to forget from the past
- second term: decide to bring how many info from the current state to the next state

- Short-term state: $h_t = o_t \otimes tanh(c_t)$, part of it is current state's info, part of it is from long-term state



Encoder-Decoder Model



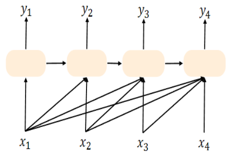
In encoder, we use RNN to generate the context, c, which is the last hidden state of the encoder. Then this c is feed into decoder, which is another RNN architecture, until a stop mark is encountered.

Attention

If we have a long sequence, the info from the far beginning state may be forgotten.
1. In each hidden state of the encoder, h_i^e, we take the dot-product of it with h_{i-1}^d (in decoder), to measure the similarity between them.
2. use softmax to calculate the weight for each dot-product attention α_{ij}
3. finally, a fixed-length context vector $c_i = \sum_j \alpha_{ij} h_j^e$ to feed into decoder.
4. Note: if we have a sentence with a length of N, then we have N fixed-length context vectors

Transformer

Self-Attention

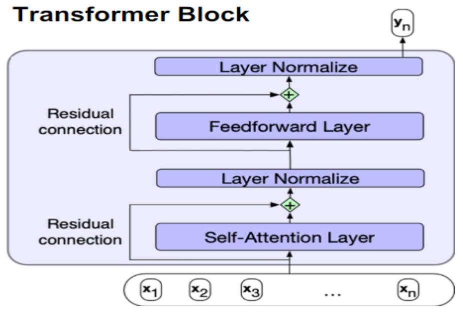


1. Compare an item of interest to a collection of other items in a way that reveals their relevance in the current context: $score(x_i, x_j) = x_i \cdot x_j$, x_i is the query, x_j is the key. x_j are all the preceding tokens.
2. A normalization of those scores to provide a probability distribution:
$\alpha_{ij} = softmax(score(x_i, x_j))$
3. A weighted sum using this distribution:
$y_i = \sum_{j \leq i} \alpha_{ij} x_j$. y_i is the value

1. Query: as the current focus of attention when being compared to all of the other preceding inputs
2. Key: In its role as a preceding input being compared to the current focus of attention
3. Value: as a value used to compute the output for the current focus of attention

1. $score(x_i, x_j) = \frac{q_i \cdot k_j}{\sqrt{d_k}}$
2. $\alpha_{ij} = softmax(score(x_i, x_j))$
3. $y_i = \sum_{j \leq i} \alpha_{ij} v_j$

Transformer Block



Residual connection: give a short-cut for the inputs to the second layer. Able to capture the nuances between inputs. Give more info to the next layer.

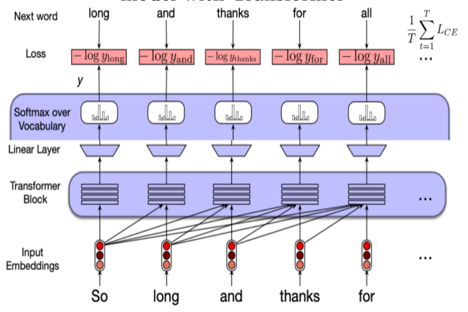
Multihead Attention

Multihead Attention located (not stack) at the same level, capture the different pattern relevance between tokens

Positional Embedding

Add position embeddings and feed it into transformer blocks together with word embeddings to capture the position of words.

Lastly, we can replace the RNN in encoder-decoder model with Transformer


Next word
long
and
thanks
for
all
Loss
$-\log y_{long}$
$-\log y_{and}$
$-\log y_{thanks}$
$-\log y_{for}$
$-\log y_{all}$
$\frac{1}{T}\sum_{t=1}^{T} L_{CE}$
y
Softmax over Vocabulary
Linear Layer
Transformer Block
Input Embeddings
So
long
and
thanks
for