Large-Scale, Distributed Machine Learning

CSE545 - Spring 2020
Stony Brook University

H. Andrew Schwartz
Supervised Learning

\[(\text{genes}) \quad X_1 \quad X_2 \quad X_3 \quad (\text{health}) \quad Y\]
Supervised Learning
Task: Determine a function, $f$ (or parameters to a function) such that $f(X) = Y$.
Supervised Learning

Task: Determine a function, $f$ (or parameters to a function) such that $f(X) = Y$
Common Goal: Generalize to new data

Model

Does the model hold up?

Original Data

New Data?
Common Goal: Generalize to new data

Does the model hold up?

Training Data

Testing Data
ML: GOAL

Training Data

Development Data

Model
Does the model hold up?

Testing Data

Model
Set training hyperparameters
N-Fold Cross Validation

Goal: Decent estimate of model accuracy
Review: Distributed ML

1. Distribute copies of entire dataset
   a. Train over all with different parameters
   b. Train different folds per worker node.

   Pro: Easy; Good for compute-bound; Con: Requires data fit in worker memories

2. Distribute data
   a. Each node finds parameters for subset of data
   b. Needs mechanism for updating parameters
      i. Centralized parameter server
      ii. Distributed All-Reduce

   Pro: Flexible to all situations; Con: Optimizing for subset is suboptimal

3. Distribute model or individual operations (e.g. matrix multiply)

   Pro: Parameters can be updated
   Con: High communication for transferring intermediar data.
update params of each node and repeat

Combine parameters

$\theta_{\text{batch0}}$

$\theta_{\text{batch1}}$
1. Linear modeling  
   (linear and logistic regression)

2. Recurrent Neural Networks  
   Where X is a sequence of data

3. Convolutional Neural Networks  
   Where X might have spatial relationships
Linear Regression: $y = wX$

Neural Network Nodes: $output = f(wX)$
From Linear Models to Neural Nets

Linear Regression: $y = wX$

Neural Network Nodes: $output = f(wX)$

Inputs  Weights  Net input function  Activation function
1  $w_0$  
$x_1$  $w_1$  
$x_2$  $w_2$  
...  $w_m$  
$x_m$  

(output) (skymind, AI Wiki)
Common Activation Functions

\[ z = wX \]

Logistic: \( \sigma(z) = \frac{1}{1 + e^{-z}} \)

Hyperbolic tangent: \( \tanh(z) = 2\sigma(2z) - 1 = \frac{e^{2z} - 1}{e^{2z} + 1} \)

Rectified linear unit (ReLU): \( \text{ReLU}(z) = \max(0, z) \)
From Linear Models to Neural Nets

Linear Regression: \( y = wX \)

Neural Network Nodes: \( output = f(wX) \)

Inputs \( 1 \) \( x_1 \) \( x_2 \) \( \ldots \) \( x_m \) \( \vdots \) \( x_m \)
Weights \( w_0 \) \( w_1 \) \( w_2 \) \( \ldots \) \( w_m \)
Net input function \( \Sigma \)
Activation function \( f \)
(output)

(skymind, AI Wiki)
From Linear Models to Neural Nets

Linear Regression: $y = wX$

Neural Network Nodes: $output = f(wX)$
Batch Normalization

**Input:** Values of $x$ over a mini-batch: $\mathcal{B} = \{x_1...m\}$;  
Parameters to be learned: $\gamma, \beta$

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

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

\[
\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \quad \text{// mini-batch variance}
\]

\[
\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \quad \text{// normalize}
\]

\[
y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \quad \text{// scale and shift}
\]

(Ioffe and Szegedy, 2015)
Batch Normalization

**Input:** Values of $x$ over a mini-batch: $\mathcal{B} = \{x_1...m\}$;  
Parameters to be learned: $\gamma, \beta$

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

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

\[
\sigma^2_\mathcal{B} \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_\mathcal{B})^2 \quad \text{// mini-batch variance}
\]

\[
\hat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma^2_\mathcal{B} + \epsilon}} \quad \text{// normalize}
\]

\[
y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \quad \text{// scale and shift}
\]

(Ioffe and Szegedy, 2015)
Batch Normalization

- $X$
- $y$

(batch_size-1)

N

N-batch_size
Batch Normalization

**Input:** Values of $x$ over a mini-batch: $\mathcal{B} = \{x_1...x_m\}$; Parameters to be learned: $\gamma, \beta$

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

- $\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i$  // mini-batch mean
- $\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2$  // mini-batch variance
- $\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}}$  // normalize
- $y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i)$  // scale and shift

---

(ioffe and Szegedy, 2015)

**Why?**
- Empirically, it works!
- Conceptually, generally good for weight optimization to keep data within a reasonable range (dividing by sigma) and such that positive weights move it up and negative down (centering).
- Small effect: When done over mini-batches, adds regularization due to differences between batches.
Feed-Forward Network

```
  Inputs   Weights
  \  \    /    \
  1  w_0  \      \    
 x_1  w_1  \  \  \  
x_2  w_2  \  \  \  
  ...  \  \  \  
x_m  w_m  \  \  \  

\[ \sum \] \[ Z \] \[ \text{activation function} \] \rightarrow \text{output}

(skymind, AI Wiki)
```

```
input layer

hidden layer 1  hidden layer 2

output layer
```
Recurrent Neural Network

Figure 9.2 Simple recurrent neural network after Elman (Elman, 1990). The hidden layer includes a recurrent connection as part of its input. That is, the activation value of the hidden layer depends on the current input as well as the activation value of the hidden layer from the previous timestep. (Jurafsky, 2019)
Backward Propagation through Time

... 
# define forward pass graph:
\[ h_{(0)} = 0 \]
for i in range(1, len(x)):
  \[ h_{(i)} = \text{tf.tanh}(\text{tf.matmul}(U, h_{(i-1)}) + \text{tf.matmul}(W, x_{(i)})) \]  # update hidden state
  \[ y_{(i)} = \text{tf.softmax}(\text{tf.matmul}(V, h_{(i)})) \]  # update output
...
\[ \text{cost} = \text{tf.reduce_mean}(-\text{tf.reduce_sum}(y*\text{tf.log}(y_{\text{pred}}))) \]
RNN: Optimization

Backward Propagation through Time

... 
define forward pass graph:
    h_{(0)} = 0
    for i in range(1, len(x)):
        h_{(i)} = tf.tanh(tf.matmul(U, x[i]) + h_{(i-1)}
    state
    y_{(i)} = tf.softmax(tf.matmul(W, h_{(i)})
... 
    cost = tf.reduce_mean(-tf.reduce_logsumexp)

To find the gradient for the overall graph, we use back propagation, which essentially chains together the gradients for each node (function) in the graph.

With many recursions, the gradients can vanish or explode (become too large or small for floating point operations).
RNN: Optimization

Backward Propagation through Time

\[ C(Y_{(2)}, Y_{(3)}, Y_{(4)}) \]

\[ W, b \]

\[ X_{(0)} \]

\[ X_{(1)} \]

\[ X_{(2)} \]

\[ X_{(3)} \]

\[ X_{(4)} \]

(Geron, 2017)
How to Addressing Vanishing Gradient?

Dominant approach: Use Long Short Term Memory Networks (LSTM)

RNN model

“unrolled” depiction

(Geron, 2017)
RNN: The GRU

Gated Recurrent Unit

(Geron, 2017)
RNN: The GRU

Gated Recurrent Unit

relevance gate

update gate

\( h_{(t-1)} \)

\( y_{(t)} \)

\( h_{(t)} \)

\( x_{(t)} \)

\( r_{(t)} \)

\( z_{(t)} \)

**GRU cell**

- Element-wise multiplication
- Addition

- logistic
- tanh

(Geron, 2017)
RNN: The GRU

Gated Recurrent Unit

relevance gate

update gate

A candidate for updating \( h \), sometimes called: \( h^\sim \)

\[ \begin{align*}
\text{GRU cell} & \quad \text{FC} \\
& \quad \text{FC} \\
& \quad \text{FC} \\
& \quad \text{FC} \\
\end{align*} \]

Element-wise multiplication

Addition

logistic

tanh

(Geron, 2017)
RNN: The GRU

Gated Recurrent Unit

\[ z(t) = \sigma(W_{xz}^T \cdot x(t) + W_{hz}^T \cdot h(t-1) + b_z) \]
\[ r(t) = \sigma(W_{xr}^T \cdot x(t) + W_{hr}^T \cdot h(t-1) + b_r) \]
\[ g(t) = \tanh(W_{xg}^T \cdot x(t) + W_{hg}^T \cdot (r(t) \otimes h(t-1)) + b_g) \]
\[ h(t) = z(t) \otimes h(t-1) + (1 - z(t)) \otimes g(t) \]

The cake, which contained candles, was eaten.
What about the gradient?

\[
\begin{align*}
    z_{(t)} &= \sigma(W_{xz}^T \cdot x_{(t)} + W_{hz}^T \cdot h_{(t-1)} + b_z) \\
    r_{(t)} &= \sigma(W_{xr}^T \cdot x_{(t)} + W_{hr}^T \cdot h_{(t-1)} + b_r) \\
    g_{(t)} &= \tanh(W_{xg}^T \cdot x_{(t)} + W_{hg}^T \cdot (r_{(t)} \otimes h_{(t-1)}) + b_g) \\
    h_{(t)} &= z_{(t)} \otimes h_{(t-1)} + (1 - z_{(t)}) \otimes g_{(t)}
\end{align*}
\]

The gates (i.e. multiplications based on a logistic) often end up keeping the hidden state exactly (or nearly exactly) as it was. Thus, for most dimensions of \( h \),

\[ h_{(t)} \approx h_{(t-1)} \]

The cake, which contained candles, was eaten.
What about the gradient?

\[ z_{(t)} = \sigma(W_{xz}^T \cdot x_{(t)} + W_{hz}^T \cdot h_{(t-1)} + b_z) \]

\[ r_{(t)} = \sigma(W_{xr}^T \cdot x_{(t)} + W_{hr}^T \cdot h_{(t-1)} + b_r) \]

\[ g_{(t)} = \tanh(W_{xg}^T \cdot x_{(t)} + W_{hg}^T \cdot (r_{(t)} \otimes h_{(t-1)}) + b_g) \]

\[ h_{(t)} = z_{(t)} \otimes h_{(t-1)} + (1 - z_{(t)}) \otimes g_{(t)} \]

The gates (i.e. multiplications based on a logistic) often end up keeping the hidden state exactly (or nearly exactly) as it was. Thus, for most dimensions of \( h \),

\[ h_{(t)} \approx h_{(t-1)} \]

This tends to keep the gradient from vanishing since the same values will be present through multiple times in backpropagation through time. (The same idea applies to LSTMs but is easier to see here).

The cake, which contained candles, was eaten.
The GRU (LSTM): Zoomed out

Take-Aways

- Simple RNNs are powerful models but they are difficult to train:
  - Just two functions $h_{(t)}$ and $y_{(t)}$ where $h_{(t)}$ is a combination of $h_{(t-1)}$ and $x_{(t)}$.
  - Exploding and vanishing gradients make training difficult to converge.

- LSTM (e.g. GRU cells) solve:
  - Hidden states pass from one time-step to the next, allow for long-distance dependencies.
  - Gates are used to keep hidden states from changing rapidly (and thus keeps gradients under control).
  - To train: mini-batch stochastic gradient descent over cross-entropy cost

(Geron, 2017)
Convolutional Neural Networks

(wikipedia)
Convolution Layer

Original image 6x6

(Barter, 2018)
Convolution Layer

Original image 6x6

(Barter, 2018)
Breakthrough in image classification: Let the model automatically learn the filter weights!
Subsampling (Pooling)

Subsampling -- reducing total grid size (i.e. reducing parameters for next layer)
Subsampling (Pooling)

Subsampling -- reducing total grid size (i.e. reducing parameters for next layer)

Types of pooling
- max
- avg
Subsampling (Pooling)

(wikipedia)

Subsampling -- reducing total grid size (i.e. reducing parameters for next layer)

Types of pooling
- max
- avg

2x2 pooling
RNN_cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(y_pred)))
    #where did this come from?

Logistic Regression Likelihood:  \[ L(\beta_0, \beta_1, ..., \beta_k|X, Y) = \prod_{i=1}^{n} p(x_i)^{y_i}(1 - p(x_i))^{1-y_i} \]

Final Cost Function:  \[ J(t) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{V} y_{i,j}^{(t)} \log \hat{y}_{i,j}^{(t)} \] -- "cross entropy error"
Standard Training Loss Function

\[
\text{RNN\_cost} = \text{tf.reduce\_mean}\left(-\text{tf.reduce\_sum}(y\times\text{tf.log}(y\_pred))\right)
#\text{where did this come from?}
\]

Logistic Regression Likelihood:
\[
L(\beta_0, \beta_1, \ldots, \beta_k | X, Y) = \prod_{i=1}^{n} p(x_i)^{y_i} (1 - p(x_i))^{1-y_i}
\]

Log Likelihood:
\[
\ell(\beta) = \sum_{i=1}^{N} y_i \log p(x_i) + (1 - y_i) \log (1 - p(x_i))
\]

Log Loss:
\[
J(\beta) = -\frac{1}{N} \sum_{i=1}^{N} y_i \log p(x_i) + (1 - y_i) \log (1 - p(x_i))
\]

Cross-Entropy Cost:
\[
J = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{|V|} y_{i,j} \log p(x_{i,j}) \quad \text{(a “multiclass” log loss)}
\]

Final Cost Function:
\[
J^{(t)} = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{|V|} y_{i,j}^{(t)} \log \hat{y}_{i,j}^{(t)} \quad \text{-- ”cross entropy error”}
\]
Review

Feed Forward Network (full-connected)
Review

Convolutional NN

(Barter, 2018)
Figure 9.2  Simple recurrent neural network after Elman (Elman, 1990). The hidden layer includes a recurrent connection as part of its input. That is, the activation value of the hidden layer depends on the current input as well as the activation value of the hidden layer from the previous timestep.  
(Jurafsky, 2019)
Can model computation (e.g. matrix operations for a single input) be parallelized?
Can model computation (e.g. matrix operations for a single input) be parallelized?

- FFN
- CNN
- RNN

- ✔️
- ✔️
Can model computation (e.g. matrix operations for a single input) be parallelized?
Ultimately limits how complex the model can be (i.e. it’s total number of parameters/weights) as compared to a CNN.

Can model computation (e.g. matrix operations for a single output) be parallelized?
The Transformer: Attention-only Models

Can handle sequences and long-distance dependencies, but....

- Don’t want complexity of LSTM/GRU cells
- Constant num edges between input steps
- Enables “interactions” (i.e. adaptations) between words
- Easy to parallelize -- don’t need sequential processing.
The Transformer: Attention-only Models

Challenge:

- Long distance dependency when translating:

\[ \text{The ball was kicked by } \text{kayla}. \]

\[ \text{Kayla kicked the ball}. \]
The Transformer: Attention-only Models

Challenge:

- Long distance dependency when translating:

  The ball was kicked by kayla.

Kayla kicked the ball.
Attention

$\alpha_{hi\rightarrow s}$

$z_1 \quad z_2 \quad z_3 \quad z_4$

values
Attention

Score function:

\[
\psi_{\text{mult}}(h_i, s) = s^T W h_i \\
\alpha_{h_i \rightarrow s} = \text{softmax}(\psi(h_i, s))
\]
Attention

Score function:

\[ \psi_{\text{mult}}(h_i, s) = s^T W h_i \]

\[ \alpha_{h_i \rightarrow s} = \text{softmax}(\psi(h_i, s)) \]

\[ c_{h_i} = \sum_{n=1}^{\lvert s \rvert} \alpha_{h_i \rightarrow s_n} z_n \]
The Transformer: Attention-only Models

Challenge:

- Long distance dependency when translating:

  Attention came about for encoder decoder models.

Then self-attention was introduced:
**Attention**

\[
\psi_{\text{mult}}(h_i, s) = s^T W h_i
\]

\[
\alpha_{h_i \to s} = \text{softmax}(\psi(h_i, s))
\]

\[
c_{h_i} = \sum_{n=1}^{\mid s \mid} \alpha_{h_i \to s_n} z_n
\]
Attention

Score function:
\[ \psi_{\text{mult}}(h_i, s) = s^T W h_i \]
\[ \alpha_{h_i \rightarrow s} = \text{softmax}(\psi(h_i, s)) \]
\[ c_{h_i} = \sum_{n=1}^{\left| s \right|} \alpha_{h_i \rightarrow s_n} z_n \]
Attention

Attention as weighting a value based on a query and key:

(Eisenstein, 2018)
The Transformer: Attention-only Models

Attention as weighting a value based on a query and key:

(Eisenstein, 2018)
The Transformer: Attention-only Models

(Eisenstein, 2018)
The Transformer: Attention-only Models

(Eisenstein, 2018)
The Transformer: Attention-only Models
The Transformer: Attention-only Models

Output

\[ FFN \]

[Diagram showing connections between variabes \( \alpha, \psi, b \) and indices \( h, w_{i-1}, w_i, w_{i+1}, w_{i+2} \)]
The Transformer: “Attention-only” models

Output

\[ y_{i-1} \quad y_i \quad y_{i+1} \quad y_{i+2} \]

\[ \alpha \quad \psi \quad h \]

\[ h_{i-1} \quad h_i \quad h_{i+1} \quad h_{i+2} \]

\[ w_{i-1} \quad w_i \quad w_{i+1} \quad w_{i+2} \]

\[ v \quad k \quad q \]
The Transformer: “Attention-only” models

Attend to all hidden states in your “neighborhood”. 

Output
The Transformer: “Attention-only” models

\[ \psi_{dp}(h_i, s) = s^T h_i \]

\[ k^q \]
The Transformer: “Attention-only” models

Output

\[
\psi \quad \psi \quad \psi \quad \psi
\]

scaling parameter

\[
\psi_{dp} (k, q) = (k^t q) \sigma
\]
The Transformer: “Attention-only” models

\[ \psi_{dp}(k, q) = (k^T q) \sigma \]

Linear layer: \( W^TX \)

One set of weights for each of \( K, Q, \) and \( V \)
The Transformer

Limitation (thus far): Can’t capture multiple types of dependencies between words.
The Transformer

Solution: Multi-head attention
Multi-head Attention
Transformer for Encoder-Decoder
Transformer for Encoder-Decoder

Stage 1
Positional Encoding

Stage 2
Multi-Head Attention

Embedding lookup

Inputs

Input Embedding

Add & Norm

Outputs

<go>

sequence index (t)

POSITIVE ENCODING

EMBEDDINGS

INPUT
je
suis
Transformer for Encoder-Decoder
Transformer for Encoder-Decoder

Stage 1: Positional Encoding
- Input Embedding
- Inputs

Stage 2: Multi-Head Attention
- Add & Norm
- Multi-Head Attention

Stage 3: Feed Forward
- Add & Norm
- Feed Forward

Residualized Connections
Transformer for Encoder-Decoder

Stage 1: Positional Encoding
- Input Embedding
- Inputs

Stage 2: Add & Norm
- Multi-Head Attention
- Feed Forward
- Add & Norm
- N x

Stage 3: Add & Norm
- Residualized Connections

Embedding lookup

With residuals

residuals enable positional information to be passed along

Without residuals
Transformer for Encoder-Decoder
Transformer for Encoder-Decoder

essentially, a language model
Transformer for Encoder-Decoder

essentially, a language model
Transformer for Encoder-Decoder

Add conditioning of the LM based on the encoder

essentially, a language model
Transformer for Encoder-Decoder

Stage 1
- Positional Encoding
- Input Embedding
- Inputs

Stage 2
- N×
- Add & Norm
- Multi-Head Attention

Stage 3
- N×
- Add & Norm
- Multi-Head Attention
- Feed Forward

Stage 4
- Add & Norm
- Multi-Head Attention
- Masked Multi-Head Attention

Stage 5
- Softmax
- Linear
- Output Probabilities

Outputs (shifted right)
Transformer (as of 2017)

“WMT-2014” Data Set. BLEU scores:

<table>
<thead>
<tr>
<th></th>
<th>EN-DE</th>
<th>EN-FR</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNMT (orig)</td>
<td>24.6</td>
<td>39.9</td>
</tr>
<tr>
<td>ConvSeq2Seq</td>
<td>25.2</td>
<td>40.5</td>
</tr>
<tr>
<td>Transformer*</td>
<td><strong>28.4</strong></td>
<td><strong>41.8</strong></td>
</tr>
</tbody>
</table>
Transformer

- Utilize Self-Attention
- Simple att scoring function (dot product, scaled)
- Added linear layers for Q, K, and V
- Multi-head attention
- Added positional encoding
- Added residual connection
- Simulate decoding by masking

https://4.bp.blogspot.com/-QIrV-PAtEkQ/W3RkOJCBlal/AAAAAAAADOg/gNZXo_eK3mMQOflfsuyPzrRfNh3gPDwLw/CLcB
GAs/s640/image1.gif
Transformer

Why?
- Don’t need complexity of LSTM/GRU cells
- Constant num edges between words (or input steps)
- Enables “interactions” (i.e. adaptations) between words
- Easy to parallelize -- don’t need sequential processing.

Drawbacks:
- Only unidirectional by default
- Only a “single-hop” relationship per layer (multiple layers to capture multiple)
BERT

Bidirectional Encoder Representations from Transformers

Produces contextualized embeddings
(or pre-trained contextualized encoder)

Drawbacks of Vanilla Transformers:
- Only unidirectional by default
- Only a “single-hop” relationship per layer
  (multiple layers to capture multiple)
BERT

Bidirectional Encoder Representations from Transformers

Produces contextualized embeddings
(or pre-trained contextualized encoder)

- Bidirectional context by “masking” in the middle
- A lot of layers, hidden states, attention heads.

Drawbacks of Vanilla Transformers:

- Only unidirectional by default
- Only a “single-hop” relationship per layer
  (multiple layers to capture multiple)
**BERT**

Sentence A = The man went to the store.
Sentence B = He bought a gallon of milk.
Label = IsNextSentence

Sentence A = The man went to the store.
Sentence B = Penguins are flightless.
Label = NotNextSentence

---

<table>
<thead>
<tr>
<th>Input</th>
<th>[CLS]</th>
<th>my</th>
<th>dog</th>
<th>is</th>
<th>cute</th>
<th>[SEP]</th>
<th>he</th>
<th>likes</th>
<th>play</th>
<th>#ing</th>
<th>[SEP]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Token Embeddings</td>
<td>$E_{[CLS]}$</td>
<td>$E_{my}$</td>
<td>$E_{dog}$</td>
<td>$E_{is}$</td>
<td>$E_{cute}$</td>
<td>$E_{[SEP]}$</td>
<td>$E_{he}$</td>
<td>$E_{likes}$</td>
<td>$E_{play}$</td>
<td>$E_{#ing}$</td>
<td>$E_{[SEP]}$</td>
</tr>
<tr>
<td>Sentence Embedding</td>
<td>$E_A$</td>
<td>$E_A$</td>
<td>$E_A$</td>
<td>$E_A$</td>
<td>$E_A$</td>
<td>$E_A$</td>
<td>$E_B$</td>
<td>$E_B$</td>
<td>$E_B$</td>
<td>$E_B$</td>
<td>$E_B$</td>
</tr>
<tr>
<td>Transformer Positional Embedding</td>
<td>$E_0$</td>
<td>$E_1$</td>
<td>$E_2$</td>
<td>$E_3$</td>
<td>$E_4$</td>
<td>$E_5$</td>
<td>$E_6$</td>
<td>$E_7$</td>
<td>$E_8$</td>
<td>$E_9$</td>
<td>$E_{10}$</td>
</tr>
</tbody>
</table>

(Devlin et al., 2019)
Bert: Attention by Layers

https://colab.research.google.com/drive/1vI0J1lhdujVjfH857hvYK1dKPTD9Kld8

(Vig, 2019)
BERT Performance: e.g. Question Answering

GLUE scores evolution over 2018-2019

- Single generic models
- 2018 Task-specific-SOTA
- Human performance

BILSTM+ELMo: 71
GPT: -75.2
BERT: 79.6
BERT Big: 81.2
BigBird: 82.2

https://rajpurkar.github.io/SQuAD-explorer/
BERT: Pre-training; Fine-tuning

Classification Layer: Fully-connected layer + GELU + Norm

Transformer encoder
12 or 24 layers
BERT: Pre-training; Fine-tuning

Transformor encoder
12 or 24 layers
BERT: Pre-training; Fine-tuning

Novel classifier
(e.g. sentiment classifier; stance detector...etc..)

Transformer encoder
12 or 24 layers

Embedding to vocab softmax
Summary

- Goal is accurate prediction of $y$ (outcome) given features ($x$)
- Use L1 or L2 penalization (as a regularization) to avoid overfit
- Reason for Train, Dev, Test split
- Components of a neural network
- Batch Normalization
- Distribution options: why is data parallelism preferred?
- Recurrent Neural Network
- Convolution Operation with Filters