D001 Economic Analysis of Non-Standard Data

Benjamin W. Arold

8. Embedding Sequences with Attention

Outline

Neural Nets: Basics

- Neural networks $\leftrightarrow$ deep learning models
  - solve machine learning problems, just like logistic regression or gradient boosted machines
  - use tensorflow, torch, or huggingface, rather than sklearn or xgboost.

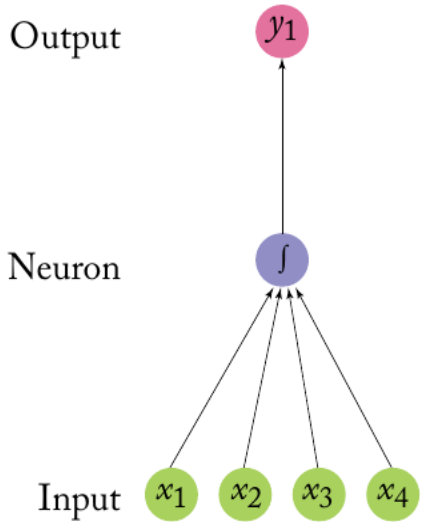
Neural Nets: Basics

- Neural networks $\leftrightarrow$ deep learning models
  - solve machine learning problems, just like logistic regression or gradient boosted machines
  - use tensorflow, torch, or huggingface, rather than sklearn or xgboost.
- **why use neural nets?**
  - greatly outperform standard ML techniques on specialized problems, for example language modeling

Neural Nets: Basics

- Neural networks $\leftrightarrow$ deep learning models
  - solve machine learning problems, just like logistic regression or gradient boosted machines
  - use tensorflow, torch, or huggingface, rather than sklearn or xgboost.
- **why use neural nets?**
  - greatly outperform standard ML techniques on specialized problems, for example language modeling
- **why not use neural nets?**
  - usually worse than standard ML on standard problems
  - outputs are a black box and difficult to interpret
  - models are often more challenging/labor-intensive to implement
  - computational constraints: training requires specialized hardware

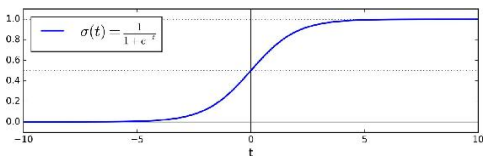
A "Neuron"



- applies dot product to vector of numerical inputs:
  - multiplies each input by a learned weight (parameter or coefficient)
  - sums these products
- applies a non-linear "activation function" to the sum
  - (e.g., the $\int$ shape indicates a sigmoid transformation)
- passes the output.

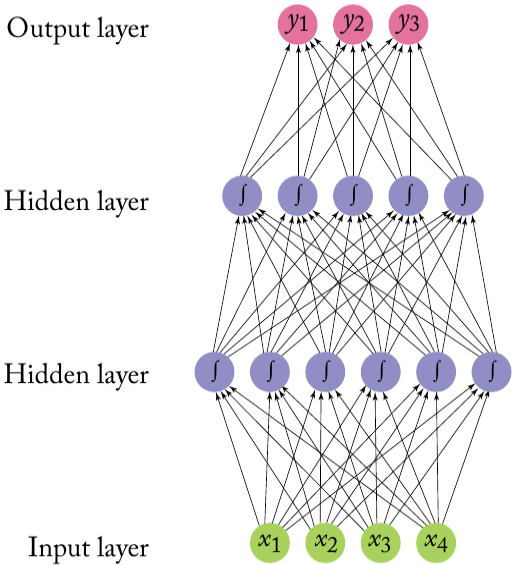
Logistic Regression ≈ "Neuron"

$$\hat{y} = \text{sigmoid}(\boldsymbol{x} \cdot \theta) = \frac{1}{1 + \exp(-\boldsymbol{x} \cdot \theta)}$$



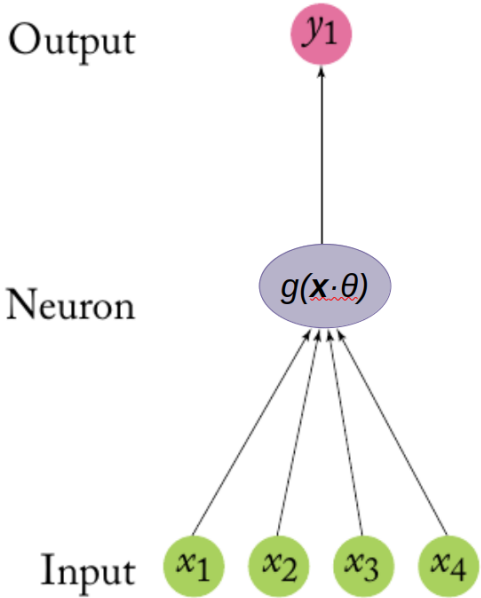
- applies dot product to vector of numerical inputs:
  - multiplies each input by a learned weight (parameter or coefficient)
  - sums these products
- applies a non-linear "activation function" (sigmoid) to the sum
- passes the output.

Multi-Layer Perceptron (MLP)



- A multilayer perceptron (also called a feed-forward network or sequential model) stacks neurons horizontally and vertically.
- alternatively, think of it as a stacked ensemble of logistic regression models.
- this vertical stacking is the "deep" in "deep learning"!

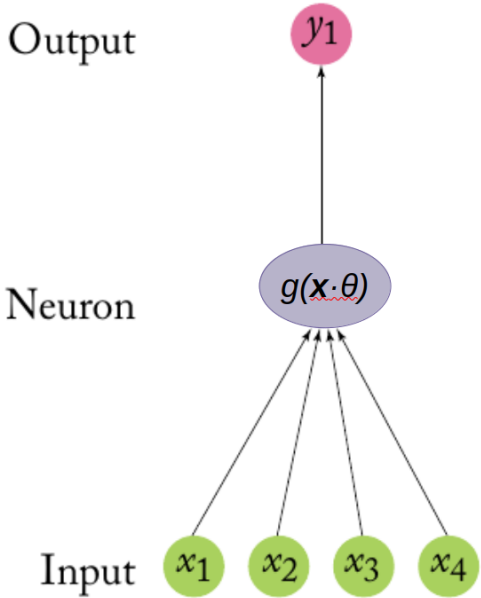
Activation functions $g(\boldsymbol{x} \cdot \theta)$



Previously we had

$$g(\boldsymbol{x} \cdot \theta) = \text{sigmoid}(\boldsymbol{x} \cdot \theta) = \frac{1}{1 + \exp(-\boldsymbol{x} \cdot \theta)}$$

Activation functions $g(\boldsymbol{x} \cdot \theta)$

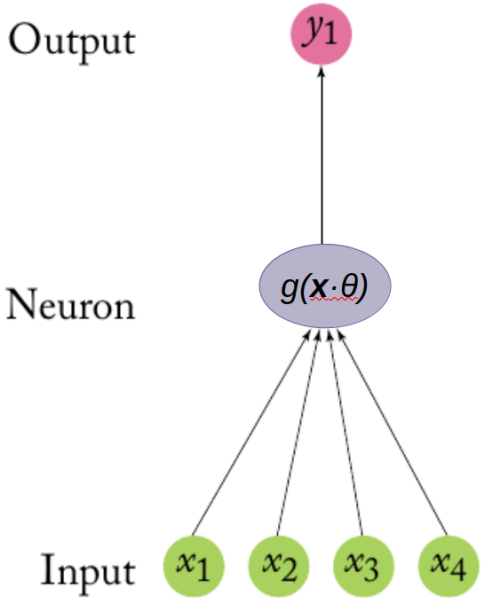


Previously we had

$$g(\boldsymbol{x} \cdot \theta) = \text{sigmoid}(\boldsymbol{x} \cdot \theta) = \frac{1}{1 + \exp(-\boldsymbol{x} \cdot \theta)}$$

But – it turns out that sigmoid does not work well in hidden layers, mainly because gradient is flat except around zero.

Activation functions $g(\boldsymbol{x} \cdot \theta)$



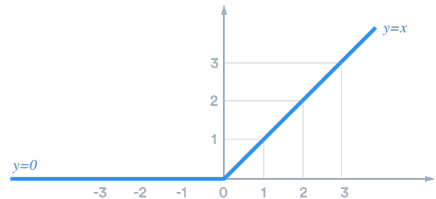
Previously we had

$$g(\boldsymbol{x} \cdot \theta) = \text{sigmoid}(\boldsymbol{x} \cdot \theta) = \frac{1}{1 + \exp(-\boldsymbol{x} \cdot \theta)}$$

But – it turns out that sigmoid does not work well in hidden layers, mainly because gradient is flat except around zero.

ReLU (rectified linear unit) function:

$$g(\boldsymbol{x} \cdot \theta) = \text{ReLU}(\boldsymbol{x} \cdot \theta) = \max\{0, \boldsymbol{x} \cdot \theta\}$$



Equation Notation: Multi-Layer Perceptron

- An multi-layer perceptron (MLP) with two hidden layers is

$$\boldsymbol{y} = \boldsymbol{g}_2(\boldsymbol{g}_1(\boldsymbol{x} \cdot \boldsymbol{\omega}_1) \cdot \boldsymbol{\omega}_2) \cdot \boldsymbol{\omega}_y$$

$$\boldsymbol{y} \in \{0,1\}^{n_y}, \boldsymbol{x} \in \mathbb{R}^{n_x}, \boldsymbol{\omega}_1 \in \mathbb{R}^{n_x \times n_1}, \boldsymbol{\omega}_2 \in \mathbb{R}^{n_1 \times n_2}, \boldsymbol{\omega}_y \in \mathbb{R}^{n_2 \times n_y}$$

  - n_1, n_2 = dimensionality in first and second hidden layer.
  - $\boldsymbol{\omega}_1, \boldsymbol{\omega}_2, \boldsymbol{\omega}_y$ = set of learnable weights for the first hidden, second hidden, and output layer
  - $\boldsymbol{g}_1(\cdot), \boldsymbol{g}_2(\cdot)$ = element-wise non-linear functions (typically ReLU) for first and second layer.

Equation Notation: Multi-Layer Perceptron

- An multi-layer perceptron (MLP) with two hidden layers is

$$\boldsymbol{y} = \boldsymbol{g}_2(\boldsymbol{g}_1(\boldsymbol{x} \cdot \boldsymbol{\omega}_1) \cdot \boldsymbol{\omega}_2) \cdot \boldsymbol{\omega}_y$$

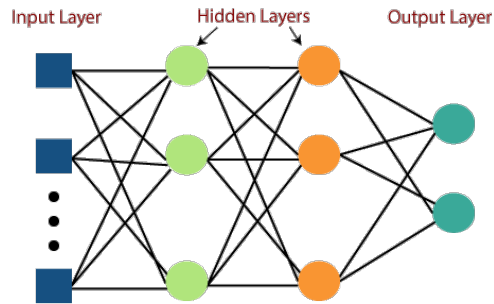
$$\boldsymbol{y} \in \{0,1\}^{n_y}, \boldsymbol{x} \in \mathbb{R}^{n_x}, \boldsymbol{\omega}_1 \in \mathbb{R}^{n_x \times n_1}, \boldsymbol{\omega}_2 \in \mathbb{R}^{n_1 \times n_2}, \boldsymbol{\omega}_y \in \mathbb{R}^{n_2 \times n_y}$$

  - n_1, n_2 = dimensionality in first and second hidden layer.
  - $\boldsymbol{\omega}_1, \boldsymbol{\omega}_2, \boldsymbol{\omega}_y$ = set of learnable weights for the first hidden, second hidden, and output layer
  - $\boldsymbol{g}_1(\cdot), \boldsymbol{g}_2(\cdot)$ = element-wise non-linear functions (typically ReLU) for first and second layer.

- Can also be written in decomposed notation:

$$\boldsymbol{h}_1 = \boldsymbol{g}_1(\boldsymbol{x} \cdot \boldsymbol{\omega}_1)$$
$$\boldsymbol{h}_2 = \boldsymbol{g}_2(\boldsymbol{h}_1 \cdot \boldsymbol{\omega}_2)$$
$$\boldsymbol{y} = \boldsymbol{h}_2 \cdot \boldsymbol{\omega}_y$$

where $\boldsymbol{h}_l$ indicate hidden layers.



Steps of NN Training (1/2)

- **Initialize Parameters:** Set weights and biases for all layers, typically randomly.
- **Input Forward Propagation:**
  - Pass input through (fully connected) layers. (For image analysis: CNNs with convolutional layers)
  - Apply activation functions (e.g., ReLU, Softmax).
- **Loss Calculation:** Compare predictions with true labels using a loss function (e.g., cross-entropy).

Steps of NN Training (2/2)

- **Backward Propagation:**
  - Compute gradients of the loss with respect to weights and biases using backpropagation.
  - Update weights through all layers via chain rule (from output to input).
- **Weight Update:** Adjust weights and biases using optimization algorithms (e.g., stochastic gradient descent).
- **Repeat:**
  - Iterate over the dataset for multiple epochs.
  - Evaluate on validation data to monitor performance and prevent overfitting.
- **Stop:** End training based on stopping criteria (e.g., convergence, maximum epochs).

Neural nets come with many hyperparameters

- **Epochs/iterations:** Number of times the model trains over the full set of images (e.g., 100 epochs)
- **Learning rate:** How much weights/coefficients change in optimization (e.g., 0.01)
- **Dropout rate:** To avoid overfitting; share of weights set to 0 (e.g., 0.5)
- **Train-validation ratio:** Split of images into sets (e.g., 80-20 split)
- **Loss function:** Evaluates model accuracy (e.g., Mean Squared Error)
- **Activation functions:** Nonlinear transformations (e.g., ReLU)
- More hyperparameters can be set (batch size etc.)

Example: NN for predicting outcomes (1/3)

- Show low-dimensional toy example predicting outcomes (vote shares) using county characteristics; can easily be adapted to high dimensional text case

County	White (%)	College (%)	Median Income (per capita, in hundreds USD)	Vote Share
Lake	87.70	16.20	215.37	0.46
Shasta	88.50	18.80	236.70	0.48
Mendocino	86.30	22.00	233.06	0.36
Sonoma	87.40	32.20	328.35	0.51
Sutter	74.00	18.70	236.02	0.55
Amador	90.70	19.30	273.47	0.52
Napa	84.80	31.30	347.95	0.60

Figure: Vote share for Hillary Clinton in the 2016 Democratic primaries in seven California counties (Webb Williams et al., 2020)

Example: NN for predicting outcomes (2/3)

$$\underset{\text{(Input Layer: } 7\times 4)}{X} \times \underset{\text{(Weights: } 4\times 2)}{\beta_1} = \underset{\text{(Hidden Layer 1 nontransformed: } 7\times 2)}{Z_0} \tag{1}$$

$$X = \begin{bmatrix} 1 & 87.7 & 16.2 & 215.37 \\ 1 & 88.5 & 18.8 & 236.70 \\ 1 & 86.3 & 22.0 & 233.06 \\ 1 & 87.4 & 32.2 & 328.35 \\ 1 & 74.0 & 18.7 & 236.02 \\ 1 & 90.7 & 19.3 & 273.47 \\ 1 & 84.8 & 31.3 & 347.95 \end{bmatrix} \quad \beta_1 = \begin{bmatrix} 0.4634 & -0.3303 \\ -0.1495 & 0.1094 \\ -0.6841 & -0.0567 \\ 0.1183 & -0.3340 \end{bmatrix} \quad Z_0 = \begin{bmatrix} 1.7436 & -63.6019 \\ 2.3684 & -70.7870 \\ 0.0775 & -69.9929 \\ 4.2065 & -102.2821 \\ 4.5243 & -72.1398 \\ 6.0468 & -82.8577 \\ 7.5294 & -109.0627 \end{bmatrix}$$

$$max(0, Z_0) = \underset{\text{(Hidden Layer 1 transformed)}}{Z_1} \tag{2}$$

$$Z_1 = \begin{bmatrix} 1.7436 & 0.0000 \\ 2.3684 & 0.0000 \\ 0.0775 & 0.0000 \\ 4.2065 & 0.0000 \\ 4.5243 & 0.0000 \\ 6.0468 & 0.0000 \\ 7.5294 & 0.0000 \end{bmatrix}$$

Figure: An artificial neural network predicting Clintons vote share in the 2016 Democratic primaries in seven California counties ($\hat{Y}$) as a function of percentage of white population (x_1), people with college education (x_2), the median income per capita (x_3), and an intermediate representation (Z_1): (1) using the input layer X to create a new hidden layer Z_0, (2) applying a nonlinear ReLu transformation to the hidden layer Z_0, ...

Example: NN for predicting outcomes (3/3)

$$\underset{\text{(Hidden Layer 1 transformed: } 7\times 3)}{Z_1} \quad \underset{\text{(Weights: } 3\times 1)}{\beta_2} \quad \underset{\text{(Output Layer nontransformed: } 7\times 1)}{\hat{Y}_0}$$

$$\begin{bmatrix} 1 & 1.7436 & 0.0000 \\ 1 & 2.3684 & 0.0000 \\ 1 & 0.0775 & 0.0000 \\ 1 & 4.2065 & 0.0000 \\ 1 & 4.5243 & 0.0000 \\ 1 & 6.0468 & 0.0000 \\ 1 & 7.5294 & 0.0000 \end{bmatrix} \times \begin{bmatrix} -0.4142 \\ 0.1084 \\ -0.1024 \end{bmatrix} = \begin{bmatrix} -0.2252 \\ -0.1575 \\ -0.4058 \\ 0.0418 \\ 0.0762 \\ 0.2412 \\ 0.4019 \end{bmatrix} \tag{3}$$

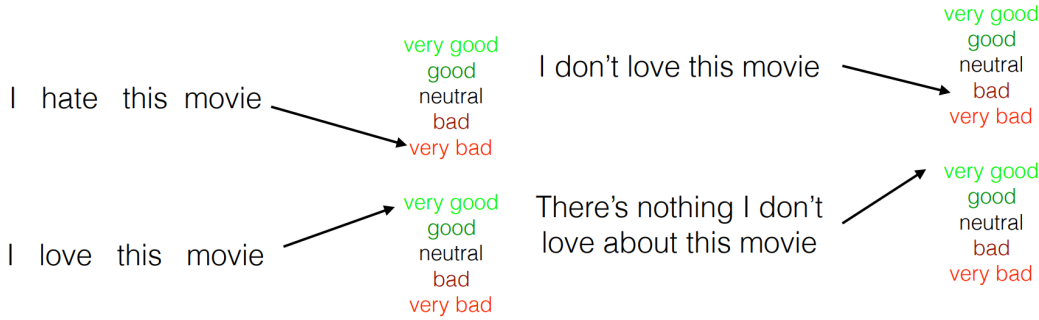
$$\underset{\text{(Output Layer transformed)}}{\hat{Y}_1}$$

$$\frac{1}{(1+e^{-\hat{Y}_0})} = \begin{bmatrix} 0.4420 \\ 0.4586 \\ 0.3979 \\ 0.5076 \\ 0.5170 \\ 0.5576 \\ 0.5962 \end{bmatrix} \tag{4}$$

Figure: ...(3) using the features in the hidden layer to generate a set of predictions ($\hat{Y}_0$), and (4) applying a sigmoid transformation to these predictions to improve model fit ($\hat{Y}_1$) (Webb Williams et al., 2020).

Outline

The Classic Sentence Classification Problem



Source: Graham Neubig slides.

- bag-of-words models won't capture the importance of "don't love" or "nothing I don't love", even with interactions / hidden layers.
- N-grams have a large feature space (especially with 4-grams) and don't share information across similar words/n-grams.

Sequence Data

- The real break-through from deep learning for NLP:
  - moving from bag-of-X representations to sequence representations.
  - Rather than inputting **counts over words x**, take as input a **sequence of tokens** $\{w_1, ..., w_t, ...w_n\}$.

Sequence Data

- The real break-through from deep learning for NLP:
  - moving from bag-of-X representations to sequence representations.
  - Rather than inputting **counts over words x**, take as input a **sequence of tokens** $\{w_1, ..., w_t, ...w_n\}$.
  - Goal: Predict the next word in a sequence given all previous words (left) or given N-1 previous words (right):

$$P(w_t \mid w_1, \ldots, w_{t-1}) \approx P(w_t \mid w_{t-N+1}, \ldots, w_{t-1})$$

Sequence Data

- The real break-through from deep learning for NLP:
  - moving from bag-of-X representations to sequence representations.
  - Rather than inputting **counts over words x**, take as input a **sequence of tokens** $\{w_1, ..., w_t, ...w_n\}$.
  - Goal: Predict the next word in a sequence given all previous words (left) or given N-1 previous words (right):

$$P(w_t \mid w_1, \ldots, w_{t-1}) \approx P(w_t \mid w_{t-N+1}, \ldots, w_{t-1})$$

  - Self-supervised ML with text (typically no pre-processing).

Sequence Data

- The real break-through from deep learning for NLP:
  - moving from bag-of-X representations to sequence representations.
  - Rather than inputting **counts over words x**, take as input a **sequence of tokens** $\{w_1, ..., w_t, ...w_n\}$.
  - Goal: Predict the next word in a sequence given all previous words (left) or given N-1 previous words (right):

$$P(w_t \mid w_1, \ldots, w_{t-1}) \approx P(w_t \mid w_{t-N+1}, \ldots, w_{t-1})$$

  - Self-supervised ML with text (typically no pre-processing).
  - Use word embeddings instead of raw word identities.

Feedforward Neural Language Model

- **Input:** One-hot encoded word vectors from a fixed-size context window.
- **Example:** Given a vocabulary size of |V|:

$$\text{dog} = [0, 0, 0, 0, 1, 0, ..., V]$$

$$\text{cat} = [0, 1, 0, 0, 0, 0, ..., V]$$

Feedforward Neural Language Model

- **Input:** One-hot encoded word vectors from a fixed-size context window.
- **Example:** Given a vocabulary size of |V|:

$$\text{dog} = [0, 0, 0, 0, 1, 0, ..., V]$$

$$\text{cat} = [0, 1, 0, 0, 0, 0, ..., V]$$

- **Embedding Vector:** Converts one-hot vectors into dense embedding vectors.

$$\mathbf{v}_{\text{dog}} = W_{\text{embed}} \times [0, 0, 0, 0, 1, 0, ...V]^T$$

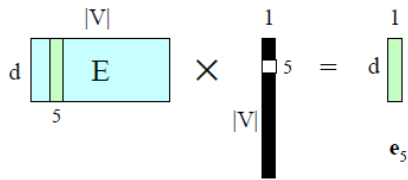


Figure: Selecting the embedding vector for word V5 by multiplying the embedding matrix E with a one-hot vector with a 1 in index 5. (Jurafsky and Martin, 2024)

Feedforward Neural Language Model

- **Embedding Layer:** The m resulting embedding vectors are concatenated to produce e, the embedding layer (where m equals the context window)
- **Hidden Layer:** Uses activation functions (ReLU, tanh, etc.).
- **Output Layer:** Produces a probability distribution over possible next words using softmax.

Feedforward Neural Language Model: Example

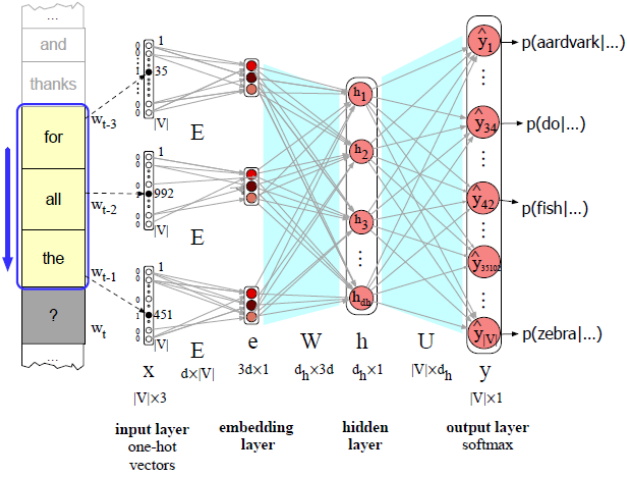


Figure: Forward inference in a feedforward neural language model. At each timestep t the network computes a d-dimensional embedding for each context word (by multiplying a one-hot vector by the embedding matrix E), and concatenates the 3 resulting embeddings to get the embedding layer e. The embedding vector e is multiplied by a weight matrix W and then an activation function is applied element-wise to produce the hidden layer h, which is then multiplied by another weight matrix U. Finally, a softmax output layer predicts at each node i the probability that the next word $\{w_t\}$ will be vocabulary word $\{V_i\}$. (Jurafsky and Martin, 2024)

Training the Neural Language Model

- **Loss function:** Cross-entropy loss for classification:

  $$L_{CE}(\hat{y}, y) = -\log \hat{y}_i = -\log p(w_t \mid w_{t-1}, \ldots, w_{t-n+1}), \quad \text{(where } i \text{ is the correct class)}$$

  where:
  - y_i is the true label (one-hot encoding).
  - $\hat{y}_i$ is the predicted probability from the softmax layer.
- **Optimization:** Uses gradient descent and backpropagation to update parameters θ (embedding matrix, weight matrices, and biases):

$$\theta_{s+1} = \theta_s - \eta \frac{\partial[-\log p(w_t \mid w_{t-1}, \ldots, w_{t-n+1})]}{\partial \theta}$$

Challenges of Feedforward Neural Language Models

- **Challenges:**
  - Computationally expensive to train and deploy.
  - Requires large datasets to perform well.
  - Limited context window (compared to RNNs, transformers).
  - Limited performance

Outline

Extensions of Feedforward Neural Language Models

Neural language models have evolved to address limitations of feedforward networks:

- **Recurrent Neural Networks (RNNs)**
  - Introduce recurrence (hidden states) to handle sequential data
  - Capture long-range dependencies, but suffer from vanishing gradients.
  - More in lecture on audio data.

Extensions of Feedforward Neural Language Models

Neural language models have evolved to address limitations of feedforward networks:

- **Recurrent Neural Networks (RNNs)**
  - Introduce recurrence (hidden states) to handle sequential data
  - Capture long-range dependencies, but suffer from vanishing gradients.
  - More in lecture on audio data.
- **Convolutional Neural Networks (CNNs)**
  - Use convolutional filters to extract local features.
  - Efficient due to parallel processing; struggles with long-range dependencies.
  - More in lecture on image data.

Extensions of Feedforward Neural Language Models

Neural language models have evolved to address limitations of feedforward networks:

- **Recurrent Neural Networks (RNNs)**
  - Introduce recurrence (hidden states) to handle sequential data
  - Capture long-range dependencies, but suffer from vanishing gradients.
  - More in lecture on audio data.
- **Convolutional Neural Networks (CNNs)**
  - Use convolutional filters to extract local features.
  - Efficient due to parallel processing; struggles with long-range dependencies.
  - More in lecture on image data.
- **Transformer Models**
  - Use self-attention for global context.
  - State-of-the-art in text data analysis
  - Examples: BERT, GPT.

Transformers

- Since a 2017 paper (Vaswani et al 2017), deep learning for NLP has been transformed by a new class of models: **transformers.**

Transformers

- Since a 2017 paper (Vaswani et al 2017), deep learning for NLP has been transformed by a new class of models: **transformers.**
- Standard approach:
  - represent documents as counts over words/phrases, shares over topics, or the average of word embeddings.

Transformers

- Since a 2017 paper (Vaswani et al 2017), deep learning for NLP has been transformed by a new class of models: **transformers.**
- Standard approach:
  - represent documents as counts over words/phrases, shares over topics, or the average of word embeddings.
- Recurrent neural nets can process whole documents word-by-word:
  - but they have to sweep through the whole document at each training epoch, so they learn too slowly.

Transformers

- Since a 2017 paper (Vaswani et al 2017), deep learning for NLP has been transformed by a new class of models: **transformers.**
- Standard approach:
  - represent documents as counts over words/phrases, shares over topics, or the average of word embeddings.
- Recurrent neural nets can process whole documents word-by-word:
  - but they have to sweep through the whole document at each training epoch, so they learn too slowly.
- Transformers overcome these limitations:
  - intuitively, they provide a way to efficiently read in an entire document and learn the meaning of all words and all interactions between words.

Self-Attention – the fundamental computation underlying transformers

- Consider a sequence of tokens with fixed length n_L, $\{w_1, ..., w_i, ..., w_{n_L}\}$
- We have word embedding vectors $x_i = E(w_i)$ with dimension n_E, producing a sequence of vectors

$$\{x_1, ..., x_i, ..., x_{n_L}\}$$

- In previous models, the sequence $x_{1:n_L}$ could be flattened to an $n_L n_E$-dimensional vector and piped to the hidden layers for use in the task, e.g. sentiment classification.

Self-Attention – the fundamental computation underlying transformers

- Consider a sequence of tokens with fixed length n_L, $\{w_1, ..., w_i, ..., w_{n_L}\}$
- We have word embedding vectors $x_i = E(w_i)$ with dimension n_E, producing a sequence of vectors

$$\{x_1, ..., x_i, ..., x_{n_L}\}$$

- In previous models, the sequence $x_{1:n_L}$ could be flattened to an $n_L n_E$-dimensional vector and piped to the hidden layers for use in the task, e.g. sentiment classification.
- A **self-attention layer** transforms $x_{1:n_L}$ into a second sequence $h_{1:n_L}$, where

$$h_i = \sum_{j=1}^{n_L} a(x_i, x_j) x_j$$

  - where $a(\cdot)$ is an attention function such that $a(\cdot) \geq 0$, $\sum a(\cdot) = 1$.
  - $\rightarrow$ each h_i becomes a weighted average of the whole sequence.
- $h_{1:n_L}$ is flattened and piped to the network's hidden layers, rather than $x_{1:n_L}$.

Basic Self-Attention

Setup:

1. Sequence of tokens $\{w_1, ..., w_i, ..., w_{n_L}\}$
2. Sequence of (trainable) embedding vectors $\{x_1, ..., x_i, ..., x_{n_L}\}$
3. Sequence of attention-transformed vectors $\{h_1, ..., h_i, ..., h_{n_L}\}$ with

$$h_i = \sum_{j=1}^{n_L} a(x_i, x_j) x_j$$

Basic Self-Attention

Setup:

1. Sequence of tokens $\{w_1, ..., w_i, ..., w_{n_L}\}$
2. Sequence of (trainable) embedding vectors $\{x_1, ..., x_i, ..., x_{n_L}\}$
3. Sequence of attention-transformed vectors $\{h_1, ..., h_i, ..., h_{n_L}\}$ with

$$h_i = \sum_{j=1}^{n_L} a(x_i, x_j) x_j$$

Basic self-attention specifies

$$a(x_i, x_j) = \frac{\exp(x_i \cdot x_j)}{\sum_{k=1}^{n_L} \exp(x_i \cdot x_k)}$$

- the dot-product $x_i \cdot x_j$, normalized with softmax such that $\sum_j a(\cdot) = 1$.

Basic Self-Attention

Setup:

1. Sequence of tokens $\{w_1, ..., w_i, ..., w_{n_L}\}$
2. Sequence of (trainable) embedding vectors $\{x_1, ..., x_i, ..., x_{n_L}\}$
3. Sequence of attention-transformed vectors $\{h_1, ..., h_i, ..., h_{n_L}\}$ with

$$h_i = \sum_{j=1}^{n_L} a(x_i, x_j) x_j$$

Basic self-attention specifies

$$a(x_i, x_j) = \frac{\exp(x_i \cdot x_j)}{\sum_{k=1}^{n_L} \exp(x_i \cdot x_k)}$$

- the dot-product $x_i \cdot x_j$, normalized with softmax such that $\sum_j a(\cdot) = 1$.
- Putting it together:

$$h_i = \sum_{j=1}^{n_L} \frac{\exp(x_i \cdot x_j)}{\sum_{k=1}^{n_L} \exp(x_i \cdot x_k)} x_j$$

- The basic self-attention transformation

$$h_i = \sum_{j=1}^{n_L} \frac{\exp(x_i \cdot x_j)}{\sum_{k=1}^{n_L} \exp(x_i \cdot x_k)} x_j$$

  is the foundational ingredient of transformers.

- The basic self-attention transformation

$$h_i = \sum_{j=1}^{n_L} \frac{\exp(x_i \cdot x_j)}{\sum_{k=1}^{n_L} \exp(x_i \cdot x_k)} x_j$$

  is the foundational ingredient of transformers.

Note the following simplifications:

- basic self-attention has no learnable parameters.
  - self-attention works indirectly through allowing the word embeddings to interact with each other
- basic self-attention ignores word order.

- The basic self-attention transformation

$$h_i = \sum_{j=1}^{n_L} \frac{\exp(x_i \cdot x_j)}{\sum_{k=1}^{n_L} \exp(x_i \cdot x_k)} x_j$$

  is the foundational ingredient of transformers.

Note the following simplifications:

- basic self-attention has no learnable parameters.
  - self-attention works indirectly through allowing the word embeddings to interact with each other
- basic self-attention ignores word order.

The big initial gain from transformers, relative to RNNs, came from basic self-attention.

- The successful models (e.g. BERT, GPT) do **add parameters and word order information** to $a(\cdot)$

Self-attention allows words to interact with each other

- Consider a sentence

  $$\text{the, cat, walks, on, the, street}$$

  with embeddings

  $$\boldsymbol{x}_{\text{the}}, \boldsymbol{x}_{\text{cat}}, \boldsymbol{x}_{\text{walks}}, \boldsymbol{x}_{\text{on}}, \boldsymbol{x}_{\text{the}}, \boldsymbol{x}_{\text{street}}$$

- Feeding this sentence into the self-attention layer produces

  $$\boldsymbol{h}_{\text{the}}, \boldsymbol{h}_{\text{cat}}, \boldsymbol{h}_{\text{walks}}, \boldsymbol{h}_{\text{on}}, \boldsymbol{h}_{\text{the}}, \boldsymbol{h}_{\text{street}}$$

  where $\boldsymbol{h}_i = \sum_{j=1}^{n} \frac{\exp(\boldsymbol{x}_i \cdot \boldsymbol{x}_j)}{\sum_k \exp(\boldsymbol{x}_i \cdot \boldsymbol{x}_k)} \cdot \boldsymbol{x}_j$.

Self-attention allows words to interact with each other

- Consider a sentence

  $$\text{the, cat, walks, on, the, street}$$

  with embeddings

  $$\boldsymbol{x}_{\text{the}}, \boldsymbol{x}_{\text{cat}}, \boldsymbol{x}_{\text{walks}}, \boldsymbol{x}_{\text{on}}, \boldsymbol{x}_{\text{the}}, \boldsymbol{x}_{\text{street}}$$

- Feeding this sentence into the self-attention layer produces

  $$\boldsymbol{h}_{\text{the}}, \boldsymbol{h}_{\text{cat}}, \boldsymbol{h}_{\text{walks}}, \boldsymbol{h}_{\text{on}}, \boldsymbol{h}_{\text{the}}, \boldsymbol{h}_{\text{street}}$$

  where $\boldsymbol{h}_i = \sum_{j=1}^{n} \frac{\exp(\boldsymbol{x}_i \cdot \boldsymbol{x}_j)}{\sum_k \exp(\boldsymbol{x}_i \cdot \boldsymbol{x}_k)} \cdot \boldsymbol{x}_j$.

Embedding layer will learn vectors $\boldsymbol{x}$ that tend to have **attention dot products** that contribute to the task at hand.

- For example, most transformers are pre-trained on a language modeling task (predicting a left-out word or sentence)
- in this task, stopwords like "the" will not be helpful.
  - the learned embedding $\boldsymbol{x}_{\text{the}}$ will tend to have a low or negative dot product with more informative words.

Autoregressive vs Autoencoding Language Models

- **Autoregressive models**:
  - e.g. **GPT = "Generative Pre-Trained Transformer"**:
  - pretrained on classic language modeling task: guess the next token having read all the previous ones.
  - during training, attention heads only view previous tokens, not subsequent tokens.
  - ideal for text generation.
- **Autoencoding models**
  - e.g. **BERT = "Bidirectional Encoder Representations from Transformers"**
  - pretrained by dropping/shuffling input tokens and trying to reconstruct the original sequence.
  - usually build bidirectional representations and get access to the full sequence.
  - can be fine-tuned and achieve great results on many tasks, e.g. text classification.