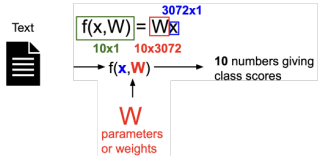
Linear Classifier

- Consider a 10 way classification problem
- 10 classes, 3072 features
- $f(x, W) = Wx$



Softmax classifier:

- Scores (vector): $s = Wx_i$
- Softmax function for probability: $P(Y = k) = \frac{e^{sk}}{\sum_j e^{sj}}$

Given: (many) input / output examples (x, y)

Find mapping f s.t. $\mathbf{y} \approx f(\mathbf{x})$

Define $L(\boldsymbol{W}) = \sum_{i=1}^{N} L\left(f_{\boldsymbol{W}}, \{\mathbf{x}_i, \mathbf{y}_i\}\right) + R(W)$

learn by optimizing $\boldsymbol{W} = \arg\min_{\boldsymbol{W}} L(\boldsymbol{W})$

How does one pick the best $\mathbf{W}$?

Sigmoid classifier (binary):

- Score (scalar): $s = \boldsymbol{w}x_i$
- Sigmoid function for probability:
  - Positive class: $P(Y = 1) = \sigma(s) = \frac{1}{1+e^{-s}}$
  - Negative class: $P(Y = 0) = 1 - \sigma(s)$

$$\frac{\partial L_i(\mathbf{W})}{\partial \mathbf{W}_{y_i}} = -x_i + p(y_i \mid x_i; \mathbf{W})x_i$$

$W_{t+1} = W_t - \alpha \nabla L(W_t)$ Gradient Descent Update Rule

where:

α: step size (learning rate)
$\nabla L(W_t)$: gradient of L at W_t
W_t; W_{t+1}: current and new value of W

Sigmoid $\sigma(x) = \frac{1}{1+e^{-x}}$

Leaky ReLU $\max(0.1x, x)$

tanh $\tanh(x)$

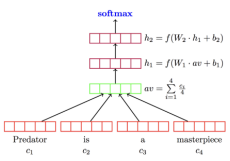
Maxout $\max(w_1^T x + b_1, w_2^T x + b_2)$

ReLU $\max(0, x)$

ELU $\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$

ReLU is a good default choice for most problems

- Deep Averaging Networks: feedforward neural network on average of word embeddings from input text



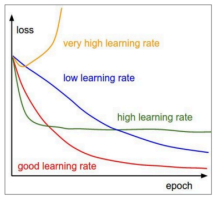
- Widely-held view: need to model syntactic structure to represent language
- DAN: Surprising that averaging can work as well as it does

$s = f(x; W_1, W_2) = W_2 g(W_1 x)$ non-linear score function

$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$ loss on predictions

$L = \frac{1}{N}\sum_{i=1}^{N} L_i + \lambda R(W_1) + \lambda R(W_2)$ Total loss: data loss + regularization

- **Learning rate**: Step size in gradient descent. Too small: slow convergence. Too large: overshoots the minimum



Batching: Training on a single example at a time is slow, batching data gives speedups due to more efficient matrix operations

- **Stochastic Gradient Descent (SGD)**: Mini-batch size of 1, noisy updates
- **Mini-Batch Gradient Descent**:Training on a small batch of examples at a time

Batch size is a hyperparameter (32, 64, 128, ...)

Skip Gram: Learning Word Embedding (Mikolov et al. 2013)

- **Input**: a corpus of text (e.g. all of Wikipedia, News articles, Books, etc.)
- **Output**: a set of embeddings: a real-valued vector for each word in the vocabulary
- We are going to learn these by setting up a fake prediction problem: predict a word's context from that word

the dog bit the man

(word = *bit*, context = *dog*)
(word = *bit*, context = *the*)

For each position $t = 1, \ldots, T$, predict context words within a window of fixed size m, given center word w_j.

Data Likelihood $= \prod_{t=1}^{T} \prod_{\substack{m \leq j \leq m \\ j \neq 0}} P(w_{t+j} \mid w_t; \theta)$

where θ are all the parameters of the model

We can write the loss as:

$$L(\theta) = -\frac{1}{T}\sum_{t=1}^{T} \sum_{\substack{m \leq j \leq m \\ j \neq 0}} \log P(w_{t+j} \mid w_t; \theta)$$

For a center word c and a context word o:

$$P(o \mid c) = \frac{\exp\left(u_o^T v_c\right)}{\sum_{w \in V} \exp\left(u_w^T v_c\right)}$$

$$\frac{\partial}{\partial v_c} \log\left(p(o \mid c)\right) = u_o - \frac{1}{\sum_{w=1}^{V} \exp\left(u_w^T v_c\right)} \cdot \sum_{x=1}^{V} \exp\left(u_x^T v_c\right) u_x$$

Distribute term across the sum:

$$= u_o - \sum_{x=1}^{V} \frac{\exp\left(u_x^T v_c\right)}{\sum_{w=1}^{V} \exp\left(u_w^T v_c\right)} u_x$$

$$= u_o - \sum_{x=1}^{V} P(x \mid c) u_x$$

= observed context vector − expected context vector

Thus center word is pulled towards words that are observed in its context, and away from those that are not. i.e
$v_c^{\text{new}} = v_c^{\text{old}} +$ observed − expected

Key Limitation of Word2Vec: Capturing cooccurrences inefficiently

- Go through each word of the whole corpus
- Predict surrounding words of each (window's center) word
- This captures cooccurrence of words **one at a time**
- **Glove**: why not capture cooccurrence counts directly?

Glove loss function:

$$L(\theta) = \frac{1}{2}\sum_{i,j=1}^{\mathcal{V}} f\left(X_{ij}\right)\left(u_i^T v_j - \log X_{ij}\right)^2$$

$\mathcal{V}$: vocabulary size
X_{ij}: cooccurrence count of word i and j
u_i, v_j: word vectors
$f(X_{ij})$: weighting function to reduce the influence of very frequent or very rare word pairs

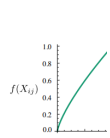


Figure:

Token: basic processing unit in NLP models

Tokenization is the process of breaking up text into tokens (words, or subwords, etc.), which then are converted to IDs through a look-up table.

Subword tokenization: break words into multiple **word pieces**

- Generally want more frequent words to be represented by fewer tokens, e.g, "the" should be a single token
- Can handle **rare** words better than word-level tokenization
- Can **share parameters** between related words (e.g., "open", "opened", "opens", "opening")
- Reduce vocabulary size → **reduce num parameters, compute+memory**

Developed for machine translation by Sennrich et al., ACL 201, Later used in BERT, T5, RoBERTa, GPT, etc.

Relies on a simple algorithm called Byte Pair Encoding (Gage, 1994)

- Iteratively merges the most frequent pair of consecutive characters, until a fixed vocabulary size is reached

Initialize vocabulary with all characters in the training data
While vocab size < max vocab size:
- Count frequency of all character pairs
- Merge most frequent pair
- Update vocabulary

Language Modeling is the task of predicting what word comes next

$$P(X_1 = x_1, X_2 = x_2, \ldots X_n = x_n)$$

$$\overset{\text{assumption}}{=} P(X_1 = x_1) \times P(X_2 = x_2 | X_1 = x_1) \times \prod_{i=3}^{n} P(X_i = x_i | X_{i-2} = x_{i-2}, X_{i-1} = x_{i-1})$$

$$= \prod_{i=1}^{n} P(X_i = x_i | X_{i-2} = x_{i-2}, X_{i-1} = x_{i-1})$$

A natural estimate (the "maximum likelihood estimate") derived from "training data":

$$q(w_i | w_{i-2}, w_{i-1}) = \frac{\text{count}(w_{i-2}, w_{i-1}, w_i)}{\text{count}(w_{i-2}, w_{i-1})}$$

$$q(laughs | the, dog) = \frac{\text{count}(the, dog, laughs)}{\text{count}(the, dog)}$$

Probability of $x_{1:m}$ is $\prod_{i=1}^{m} p(x_i)$

Log-probability of $x_{1:m}$ is $\sum_{i=1}^{m} \log_2 p(x_i)$

Average log-probability per word of $x_{1:m}$ is:

$$l = \frac{1}{M}\sum_{i=1}^{m} \log_2 p(x_i)$$

where $M = \sum_{i=1}^{m} |x_i|$ (number of words in the corpus)

Perplexity (relative to $x_{1:m}$) is 2^{-l}
Lower is better.

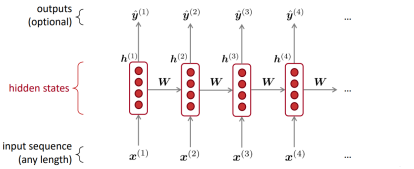
Unrolled Recurrent Neural Networks (RNNs)

hidden states

$$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_e \boldsymbol{e}^{(t)} + \boldsymbol{b}_1\right)$$

$\boldsymbol{h}^{(0)}$ is the initial hidden state

Core idea: Apply the same weights $\boldsymbol{W}$ repeatedly



$$\frac{1}{T}\sum_{t=1}^{T} -\log \hat{\boldsymbol{y}}_{\boldsymbol{x}_{t+1}}^{(t)}$$

What probability did we give to the actual next word

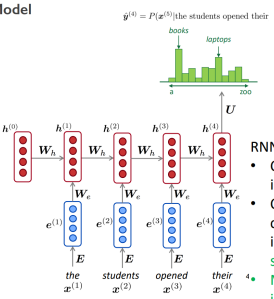
RNN Language Model

output distribution
$\hat{\boldsymbol{y}}^{(t)} = \text{softmax}\left(\boldsymbol{U}\boldsymbol{h}^{(t)} + \boldsymbol{b}_2\right) \in \mathbb{R}^{|V|}$

hidden states
$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_e \boldsymbol{e}^{(t)} + \boldsymbol{b}_1\right)$
$\boldsymbol{h}^{(0)}$ is the initial hidden state

word embeddings
$\boldsymbol{e}^{(t)} = \boldsymbol{E}\boldsymbol{x}^{(t)}$

words / one-hot vectors
$\boldsymbol{x}^{(t)} \in \mathbb{R}^{|V|}$



$$J(\theta) = \frac{1}{T}\sum_{t=1}^{T} J^{(t)}(\theta)$$

RNN Advantages:

- Can process any length input
- Computation for step t can (in theory) use information from many steps back
- Model size doesn't increase for longer input

RNN Disadvantages:

- Recurrent computation is slow
- In practice, difficult to access information from many steps back

Training RNNs not much different from FFN, by unrolling the RNN we effectively create a feedforward network

Key Difference is Weight Sharing in RNNs: same weight is used at every time step; in FFN, each layer has its own weights.

Gradient clipping: if the norm of the gradient is greater than some threshold, scale it down before applying SGD update

Algorithm 1 Pseudo-code for norm clipping

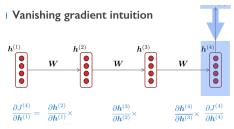
$\hat{\mathbf{g}} \leftarrow \frac{\partial \mathcal{E}}{\partial \theta}$
if $\|\hat{\mathbf{g}}\| \geq threshold$ **then**
  $\mathbf{g} \leftarrow \frac{threshold}{\|\hat{\mathbf{g}}\|}\hat{\mathbf{g}}$
end if

If the gradient becomes too big, then the SGD update step becomes too big:

$$\theta^{new} = \theta^{old} - \underbrace{\alpha}_{\text{learning rate}} \underbrace{\nabla_\theta J(\theta)}_{\text{gradient}}$$

This can cause bad updates: we take too large a step and reach a weird and bad parameter configuration (with large loss)

Vanishing gradient intuition



If numbers are < one, multiplying them together gets us a tiny product, eventually converges towards zero **(vanishing gradients)**

If numbers > one, we get the opposite problem, the product will increase exponentially in the length of the sequence **(exploding gradients)**

If FFN, layers have different weights, some numbers might be small, some large, there is a balance

Solution, a different architecture: Bidirectional RNN

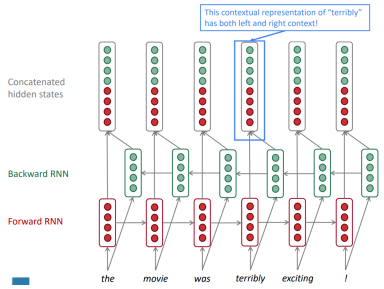
Note: bidirectional RNNs are only applicable if you have access to the entire input sequence.

- They are **not** applicable to Language Modeling, because in LM you *only* have left context available.

If you do have entire input sequence (e.g. any kind of encoding), bidirectionality is powerful (you should use it by default).

For example, BERT (**Bidirectional** Encoder Representations from Transformers) is a powerful pretrained contextual representation system built on bidirectionality.

- You will learn more about BERT later in the course!



LSTM: hidden state, cell

In step t, there is a hidden state $\boldsymbol{h}^{(t)}$ *and* a cell state $\boldsymbol{c}^{(t)}$

- Both are vectors length n
- The cell stores long-term information
- The LSTM can erase, write and read information from the cell

New cell content: this is the new content to be written to the cell

Cell state: erase ("forget") some content from last cell state, and write ("input") some new cell content

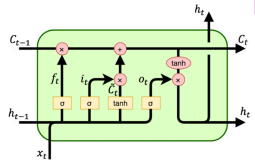
Hidden state: read ("output") some content from the cell

$$\tilde{\boldsymbol{c}}^{(t)} = \tanh\left(\boldsymbol{W}_c \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_c \boldsymbol{x}^{(t)} + \boldsymbol{b}_c\right)$$

$$\boldsymbol{c}^{(t)} = \boldsymbol{f}^{(t)} \circ \boldsymbol{c}^{(t-1)} + \boldsymbol{i}^{(t)} \circ \tilde{\boldsymbol{c}}^{(t)}$$

$$\boldsymbol{h}^{(t)} = \boldsymbol{o}^{(t)} \circ \tanh \boldsymbol{c}^{(t)}$$

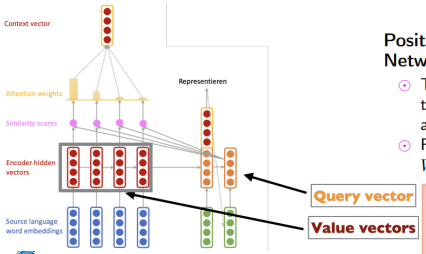
Gates are applied using element-wise product



N-gram models assessment

Pros:
- Easy to understand
- Good enough for machine translation, speech recognition, ...

Cons:
- Markov assumption is linguistically inaccurate
- Data sparseness
- "Out of vocabulary" problem



Attention is a general Deep Learning technique

We have some *values* $\boldsymbol{h}_1, \ldots, \boldsymbol{h}_N \in \mathbb{R}^{d_1}$ and a *query* $\boldsymbol{s} \in \mathbb{R}^{d_2}$

Attention always involves:

1. Computing the *attention scores* $\boldsymbol{e} \in \mathbb{R}^N$ (There are multiple ways to do this)
2. Taking softmax to get *attention distribution* α:
$$\alpha = \text{softmax}(\boldsymbol{e}) \in \mathbb{R}^N$$
3. Using attention distribution to take weighted sum of values:
$$\boldsymbol{a} = \sum_{i=1}^{N} \alpha_i \boldsymbol{h}_i \in \mathbb{R}^{d_1}$$

thus obtaining the *attention output* $\boldsymbol{a}$ (sometimes called the *context vector*)

There are several ways you can compute $\boldsymbol{e} \in \mathbb{R}^N$ from $\boldsymbol{h}_1, \ldots, \boldsymbol{h}_N \in \mathbb{R}^{d_1}$ and $\boldsymbol{s} \in \mathbb{R}^{d_2}$:

Basic dot-product attention: $\boldsymbol{e}_i = \boldsymbol{s}^T \boldsymbol{h}_i \in \mathbb{R}$
- Note: this assumes $d_1 = d_2$.

Multiplicative attention: $\boldsymbol{e}_i = \boldsymbol{s}^T \boldsymbol{W} \boldsymbol{h}_i \in \mathbb{R}$ [Luong, Pham, and Manning 2015]
- Where $\boldsymbol{W} \in \mathbb{R}^{d_2 \times d_1}$ is a weight matrix. Perhaps better called "bilinear attention"

Reduced-rank multiplicative attention: $e_i = s^T (\boldsymbol{U}^T \boldsymbol{V}) h_i = (\boldsymbol{U}s)^T (\boldsymbol{V}h_i)$
- For low rank matrices $\boldsymbol{U} \in \mathbb{R}^{k \times d_2}, \boldsymbol{V} \in \mathbb{R}^{k \times d_1}, k \ll d_1, d_2$

Additive attention: $\boldsymbol{e}_i = \boldsymbol{v}^T \tanh(\boldsymbol{W}_1 \boldsymbol{h}_i + \boldsymbol{W}_2 \boldsymbol{s}) \in \mathbb{R}$ [Bahdanau, Cho, and Bengio 2014]
- Where $\boldsymbol{W}_1 \in \mathbb{R}^{d_3 \times d_1}, \boldsymbol{W}_2 \in \mathbb{R}^{d_3 \times d_2}$ are weight matrices and $\boldsymbol{v} \in \mathbb{R}^{d_3}$ is a weight vector.
- d_3 (the attention dimensionality) is a hyperparameter
- "Additive" is a weird/bad name. It's really using a feed-forward neural net layer.

Attention significantly improves NMT performance
- It's very useful to allow decoder to focus on certain parts of the source

Attention solves the bottleneck problem
- Attention allows decoder to look directly at source; bypass bottleneck

Attention provides some interpretability
- By inspecting attention distribution, we can see what the decoder was focusing on
- We get (soft) alignment for free!
- This is cool because we never explicitly trained an alignment system
- The network just learned alignment by itself

In greedy decoding, usually we decode until the model produces an <END> token
- **For example:** <START> he hit me with a pie <END>

In beam search decoding, different hypotheses may produce <END> tokens on different timesteps
- When a hypothesis produces <END>, that hypothesis is complete.
- Place it aside and continue exploring other hypotheses via beam search.

$$\frac{1}{t} \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$$

Attention is All you Need

- Attention
  - Self-attention
  - Interspersing of Attention and Feedforward layers
  - Multiple attention heads in parallel
  - Encoder-Decoder Attention (cross-attention)
- Positional Encoding (attention is order-agnostic)
- Optimization Tricks
  - Residual network (ResNet) structure
  - LayerNorms
  - Scaled dot-product attention

Given: Query vector q and a set of Key-Value (k-v) vector pairs.

Output: a weighted sum of the values, where the weights are determined by the similarity of the query to the keys.
- A selective summary of the values, with respect to the query.

k and v are derived from the same input x:

$$k = W_k \cdot x \quad v = W_v \cdot x$$

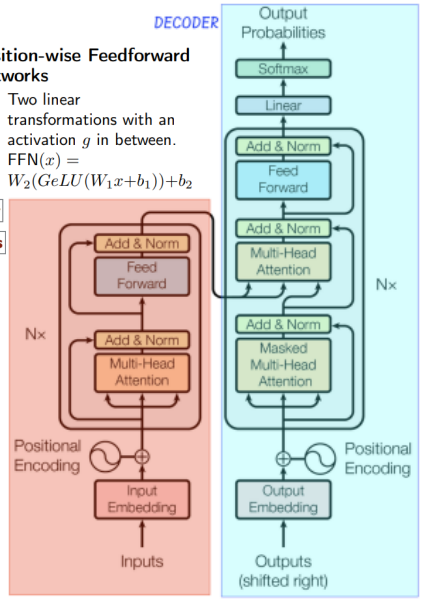
The query q can come from a separate input y:

$$q = W_q \cdot y$$

Or from the same input x, then we call it "self-attention":

$$q = W_q \cdot x$$

DECODER



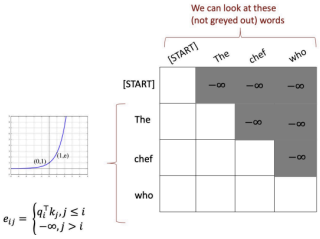
Position-wise Feedforward Networks

- Two linear transformations with an activation g in between.
- $\text{FFN}(x) = W_2(GeLU(W_1 x + b_1)) + b_2$

$$A(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{d_k}}\right) \cdot \boldsymbol{V}$$

We want to mask with upper triangular matrix. In PyTorch use: `torch.triu`

Setting **attention score** to $-\infty$ ensures that attention probability will be zero.



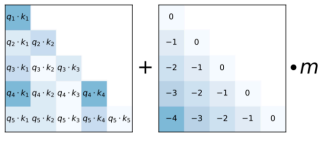
$$e_{ij} = \begin{cases} q_i^T k_j, j \le i \\ -\infty, j > i \end{cases}$$

Encoder-Decoder Attention

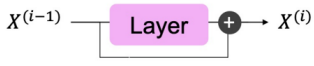
- Each decoded token can "look at" the encoder's output (as in seq2seq w/ attention)
- This allows the decoder to focus on relevant parts of the encoder's output.

Absolute vs Relative Positional Encodings

- **Absolute**: add an encoding to the input in **hope** that relative position will be captured
  - Vaswani et al. 2017: Positional Encodings
  - Fails to extrapolate to longer sequences
- **Relative**: **Explicitly** encode relative position e.g., "I" is 3 words away from "run"
  - Press et al. 2021: "Attention with Linear Biases" (ALiBi), **parameter-free** linear bias added to dot-product similarity. Extrapolates to longer sequences.



Residual connections: training converges faster, and allows for deeper networks.



Layer normalization: Normalizes the outputs to be within a consistent range, preventing too much variance in scale of outputs

Challenges with RNNs
- Gradient vanishing and explosion
  - Long-range dependencies are not captured
- Recurrence prevents parallel computation
  - Slow training

Transformer Networks
- Facilitate long range dependencies
- No gradient vanishing and explosion
- No recurrence: facilitate parallel computation

Even though most parameters and FLOPs are in feedforward layers, Transformers are still limited by quadratic $O(n^2)$ complexity of self-attention

Pretraining: The model learns general knowledge from a large, diverse dataset.
Finetuning: The model adapts this general knowledge to a specific task with a smaller, task-specific dataset.

ELMo is built using a deep **Bi-LSTM** model, where each word token's embedding is a function of:

$$\mathbf{h}_k^{LM} = [\overrightarrow{\mathbf{h}_k^{LM}}; \overleftarrow{\mathbf{h}_k^{LM}}]$$

$\overrightarrow{\mathbf{h}_k^{LM}}$ and $\overleftarrow{\mathbf{h}_k^{LM}}$ represent the hidden states of forward and backward LSTMs for word k.

$$\mathbf{ELMo}_k = \gamma \sum_{j=0}^{L} s_j \mathbf{h}_{k,j}^{LM}$$

re:
$\mathbf{h}_{k,j}^{LM}$ is the j-th layer of the LSTM for word k.
s_j: trainable scalar weights.
γ: trainable scalar parameter to scale the final embedding.

d: Nucleus Sampling

:tic Decoding Method using Top-p Vocabulary

Given a probability distribution $P(x \mid x_{1:i-1})$, the **top-p vocabulary** $V^{(p)} \subset V$ is defined as the smallest set of tokens satisfying:

$$\sum_{x \in V^{(p)}} P(x \mid x_{1:i-1}) \ge p$$

Focus on (the nucleus) ensures sampling focuses on high-probability tokens while discarding the unreliable tail

Let $p' = \sum_{x \in V^{(p)}} P(x \mid x_{1:i-1})$. Original distribution is re-scaled to a new distribution, from which the next word is sampled:

$$P'(x \mid x_{1:i-1}) = \begin{cases} P(x \mid x_{1:i-1}) / p' & \text{if } x \in V^{(p)} \\ 0 & \text{otherwise} \end{cases}$$

At each time step, top-k sampling considers only the k token with the highest probabilities, re-scaling these probabilities and sampling from them

Sampling with Temperature

High temperature: more randomness; **Low temperature**: less randomness

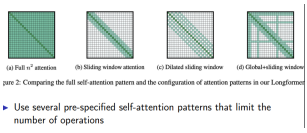
Given the scores $u_{1:|V|}$ and temperature t, adjust softmax as:

$$p(x = V_l \mid x_{1:i-1}) = \frac{\exp(u_l / t)}{\sum_{l'} \exp(u_{l'} / t)}$$

Negative Sampling
- **Purpose:**
  - Efficiently approximate the softmax function by simplifying the computation.
- **Method:**
  - For each positive example (target-context pair), sample k negative examples (words not in the context).
  - Update weights for both positive and negative samples.
- **Loss Function:**
  - Uses a binary logistic regression objective:
$$\mathcal{L} = -\left(\log \sigma(u_{w_o}^\top v_{w_i}) + \sum_{i=1}^{k} \log \sigma(-u_{w_i}^\top v_{w_i})\right)$$
  where:
  - σ is the sigmoid function.
  - u_{w_o}: Output embedding of context word.
  - v_{w_i}: Input embedding of target word.
  - u_{w_i}: Negative sample words.

Teacher Forcing During Training
- **Definition:**
  - During training, the decoder is provided with the ground truth previous token y_{i-1} as input rather than its own previous prediction.
- **Benefits:**
  - **Stability:**
    - Reduces the accumulation of errors over time steps.
  - **Faster Convergence:**
    - The model learns the correct mapping between inputs and outputs more efficiently.
- **Limitations:**
  - **Exposure Bias:**
    - At inference time, the model relies on its own predictions, which may lead to errors if it hasn't learned to handle them during training.

Alternatives:
- **Scheduled Sampling:**
  - Gradually replaces ground truth inputs with model predictions during training.



pare 2: Comparing the full self-attention pattern and the configuration of attention patterns in our Longformer

- Use several pre-specified self-attention patterns that limit the number of operations

-training BERT: Masked Language Model

Masked Language Model (MLM) is one of the two pre-training tasks in BERT.

- Given an input sentence, BERT uses the context of surrounding words to predict the masked word:
$$P(\text{word}_i \mid \text{context}) = \text{softmax}(\mathbf{W} \cdot \mathbf{h}_i)$$
- Where:
  - $\mathbf{h}_i$: The hidden state of the Transformer for word i.
  - $\mathbf{W}$: Output projection matrix.

LMs place a distribution $\text{P}(\text{x}_\text{i} \mid \text{x}_1, \ldots, \text{x}_{\text{i}-1})$

How do we generate text from these?
- **Greedy decoding**: take the word with the highest probability at each step
- **Beam search**: find the sequence with the highest probability
- **Sample from the model**: draw x_i from that distribution

LMs place a distribution $\text{P}(\text{x}_\text{i} \mid \text{x}_1, \ldots, \text{x}_{\text{i}-1})$. How do we generate text from these?

Maximization-based decoding:
- **Greedy decoding**: pick the most likely word at each step
- **Beam search**: keep track of the top k most likely sequences

Sampling-based decoding:
- **Random sampling**: sample from the full distribution
- **Nucleus sampling**: sample from the top p fraction of the distribution
- **Top-k sampling**: sample from the top k most likely words

Encoder : $P(x_i \mid x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$
- **Bidirectional attention**, trained with masked language modeling
- **To use in practice**: Ignore this probability distribution. Fine-tune the model for some other task $\text{P}(\boldsymbol{y} \mid \boldsymbol{x})$

Decoder: $P(x_i \mid x_1, \ldots, x_{i-1})$
- **Unidirectional attention**, trained to predict the next word
- Can ignore this probability distribution and train a model for $\text{P}(\boldsymbol{y} \mid \boldsymbol{x})$.
- **To use in practice**: Generate text by sampling from this distribution

Momentum:
- Incorporates past gradients to smooth updates.
- Update rule:
$$v_t = \gamma v_{t-1} + \eta \nabla_\theta \mathcal{L}(\theta)$$
$$\theta_{\text{new}} = \theta_{\text{old}} - v_t$$
where γ is the momentum coefficient.

Sinusoidal Positional Encoding:
- **Formulas:**
$$\text{PE}_{(\text{pos},2i)} = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$
$$\text{PE}_{(\text{pos},2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$
  - pos: Position in the sequence.
  - i: Dimension index.
  - d_{model}: Model dimensionality.

Properties:
- Provides unique encoding for each position.
- Allows the model to generalize to sequences longer than those seen during training.

Alternative:
- **Learned Positional Embeddings:**
  - Position embeddings are learned during training.
  - May not generalize to longer sequences.