Pre-Requisits for understanding ”Attention is all you Need” Paper, inspired from the course: ”MIT 6.S191: Introduction to Deep Learning

Mohamad Lakkis

August, 2024

Table of Contents

- **Optimization and training:**
  - SGD, Adam, ...
  - Cross-entropy loss, which is used in classification including language models

1 Introduction

We will explore different subjects and take notes

2 Linear Algebra and Probability

Eigenvectors and Eigenvalues

- A Vector that undergoes pure scaling(no rotation) when multiplied by a matrix is called an eigenvector of that matrix.
- The scaling factor is called the eigenvalue.
- $Ax = \lambda x$
- A: Matrix, x: Eigenvector, λ: Eigenvalue

PDF,CDF,PPF

- PPF is the inverse of CDF, so let's say we wanted to find the value of x that corresponds to a certain probability, we would use the PPF. SO in other words $F(x) = P(X \leq x) and F^{-1}(p) = PPF(p)$
- As for the PDF, it is the derivative of the CDF, so it gives us the gradient of the cdf at a value x. $f(x) = \frac{dF(x)}{dx}$
- x might designate say the height, and so the CDF would give us the probability of a person being shorter than x. The PDF would give us the slope of the CDF at x, so it would give us the probability of a person being between x and x+dx.

Neural Networks

Activation Functions

- Sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$, not always used because of the vanishing gradient problem because it squeezes the input.
- Tanh: $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
- ReLU: $f(x) = max(0, x)$, not always used because of the dying ReLU problem (ignoring negative values).
- Leaky ReLU: $f(x) = max(0.01x, x)$ to solve the dying ReLU problem.
- Softmax: $f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$, for classification problems. (used in the output layer). This is used instead of $\frac{x_i}{\sum x_j}$, because it guarantees that the

sum of the probabilities is 1 and they are probabilities, because the regular one might lead to values greater than 1 (i.e. not a probability) or less than 0 (i.e. not a probability). We also use it for non linearity. It is also more stable than the regular one. And also it works well with the gradient properties(it helps in reducing the likelihood of vanishing gradient).

CNN-Convolution Neural Networks

`Images/cnn.png`

Figure 1: CNN Architecture

- We do this because deep in the layer we can then ignore the spatial structure, that is why we use flatten and then treat it as a regular fully connected neural network.
- So each box that yo see in the image is a collection of neurons, and each neuron is connected to a small region of the input image. And after many

layers each neuron is connected to a small region in the filtered map of the previous layer.

- So we can see there is many applications for CNNs.
- So after the feature extraction we have so many possibilities, we can do whatever we want with the features, we can use them for classification, detection, segmentation, etc.

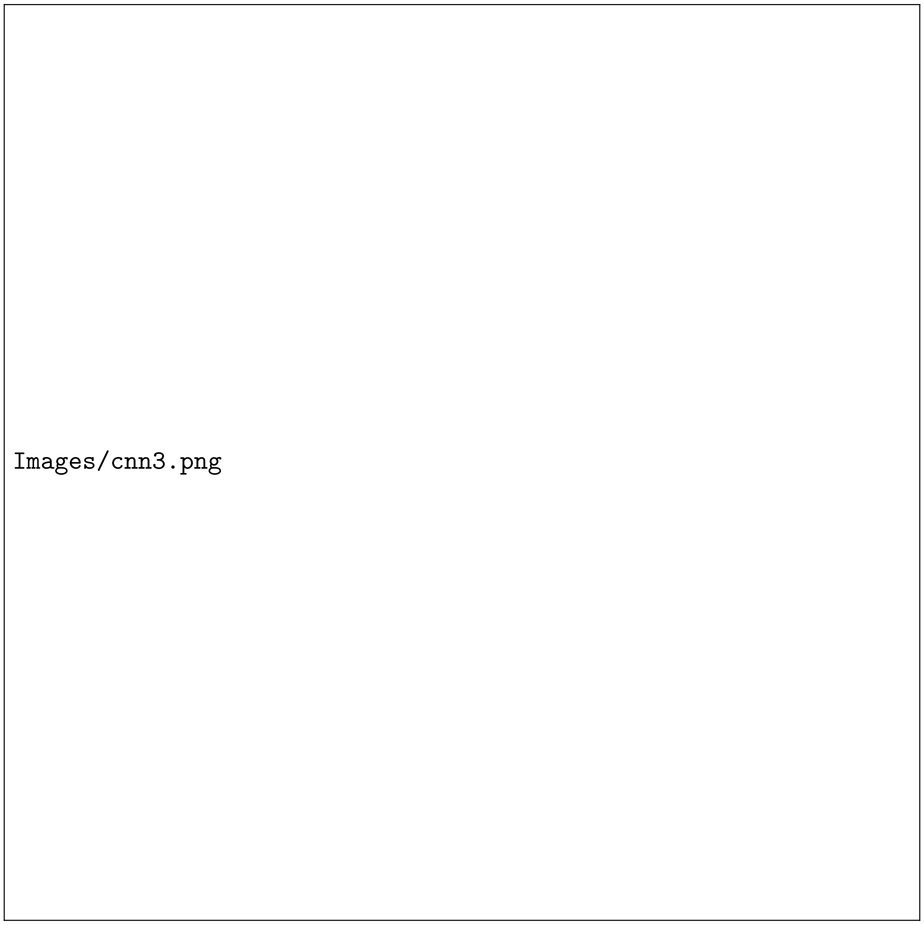

Figure 2: CNN example of self-driving car

- F.c. means a fully connected network.

Images/cnn2.png

Figure 3: CNN Code Sample

How to actually solve the optimization problem? - Gradient Descent

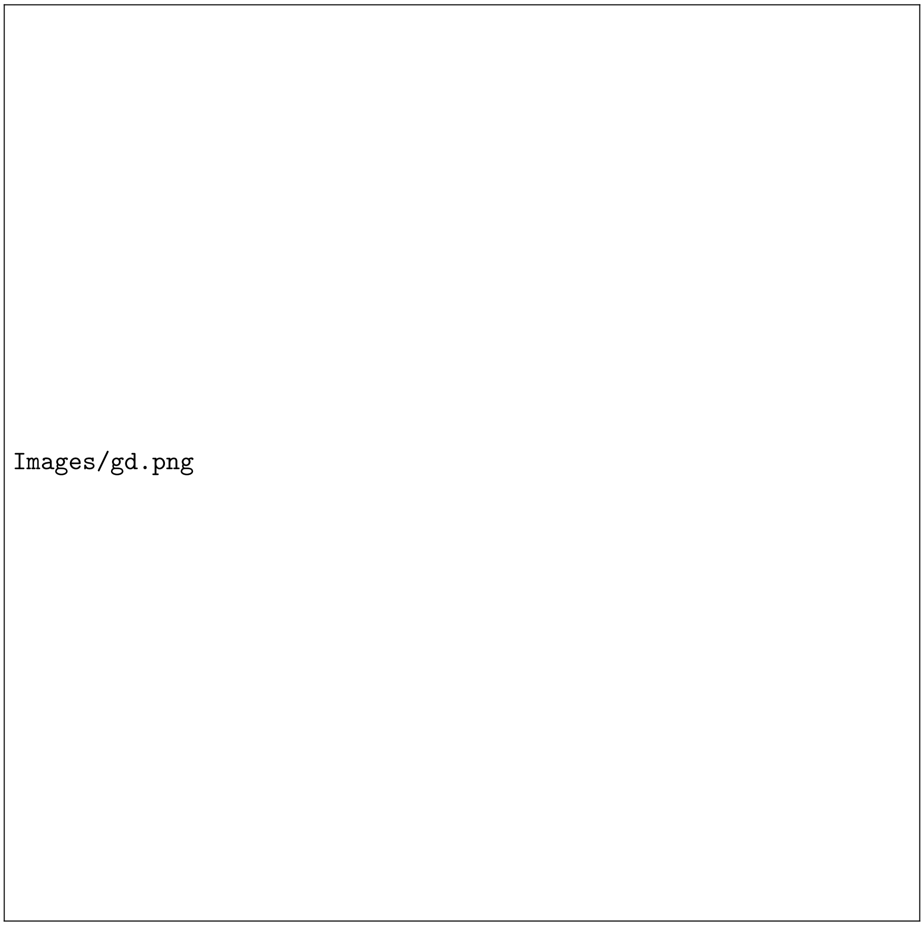

Figure 4: Gradient Descent

- We can see that the learning rate is very important, if it is too small it will take a long time to converge, if it is too big it might overshoot the minimum.
- And we can see that if we choose a wrong initial value or a wrong learning rate we might get stuck in a local minimum.

To compute the gradient we use the chain rule(backpropagation)

Images/backprop.png

Figure 5: Backpropagation

- You might be asking why we use backpropagation, why not just compute $\frac{\partial J(W)}{\partial w_i}$? Because the function $J(W)$ is a composition of functions, so we need to use the chain rule to compute the gradient. We can't directly compute the gradient with respect to w_i, instead, it's a complex function involving multiple layers of computation. Backpropagation, using the chain rule, allows us to correctly and efficiently compute this gradient by taking into account the entire chain of dependencies from w_1 to $J(W)$. (i.e. $J(W) \rightarrow \hat{y} \rightarrow z_1 \rightarrow w_1$)

In the end it all depends on the learning rate, so how do we choose this learning rate? One idea is to work with it as a hyperparameter, and try different values and see which one works best. Another idea is to use adaptive learning rates!!

Adaptive Learning Rates - Optimizing Gradient Descent

- Adagrad: It adapts the learning rate to the parameters, so it gives a different learning rate to each parameter. It is good for sparse data.
- RMSprop: It is similar to Adagrad, but it divides the learning rate by the square root of the sum of the squares of the gradients. It is good for non-stationary data.
- Adam: It is a combination of RMSprop and momentum. It is good for large datasets and high-dimensional data.
- SGD: Stochastic Gradient Descent, it is the most basic one, it is good for large datasets. (we chose a random batch (sample of the data) and we compute the gradient on this batch and we update the weights, and we repeat this process until we reach the minimum).

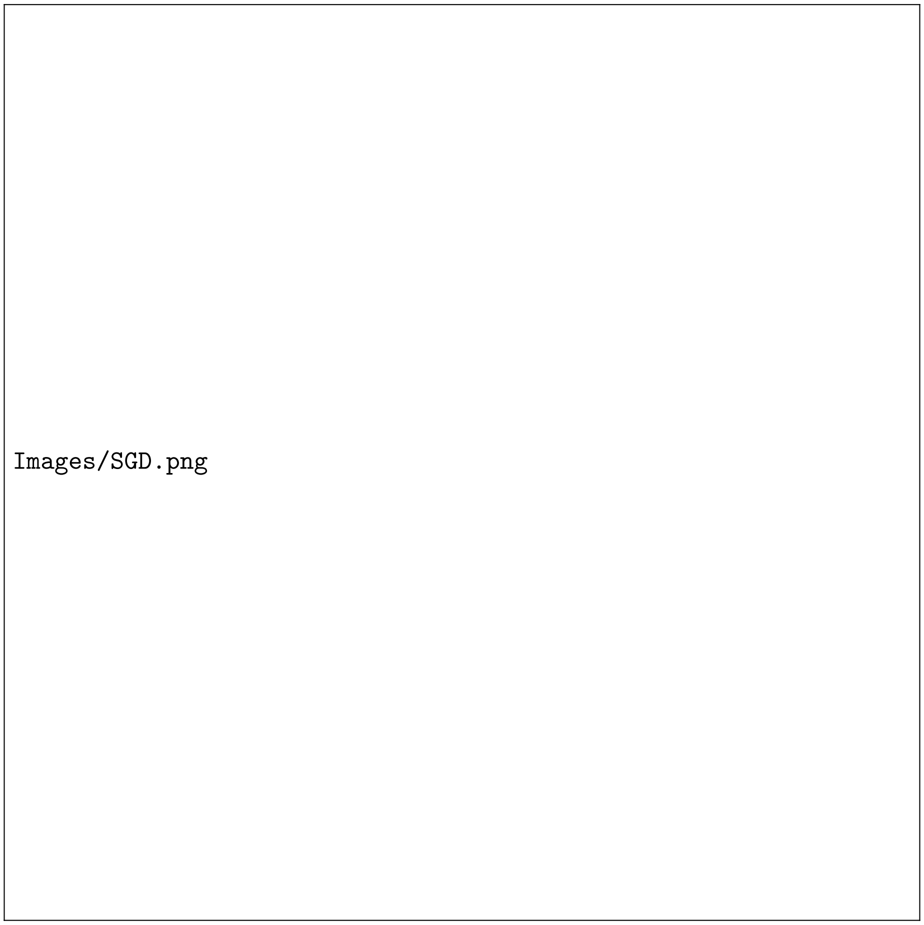

Figure 6: SGD

How to apply regularization?

2.0.1 Dropout

- During training set some activations to 0, typically 50% of the activations are set to 0, which forces the network to not rely on a single neuron. $tf.keras.layers.Dropout(p = 0.5)$

2.0.2 Early Stopping

- We stop training when the test error starts to increase, because this means that the model is overfitting. (find the sweet spot between underfitting and overfitting)

Sequence Models

RNN - Recurrent Neural Networks

Images/rnn.png

Figure 7: RNN

- RNNs is the foundation of sequence models.
- They are used for sequence data, like time series, text, etc.
- The figure above help us understand the Architecture of an RNN.
- So what you see is that on the right this is ultimately what we are doing, but we don't have many models, so think about it as a recursion, so it is re going back to itself on the next input, so it is like a loop. Which we can see from the diagram on the left. This means that at time equal t we are

using the information from time = t-1, modeled by h_{t-1}, and the input x_t at this particular time.

- Now let's formalize this a bit more, not that in the figure below we are modifying the weights as we go along the sequence, so we are not using the same weights for all the inputs, we are using the same weights but we are modifying them as we go along the sequence.

Images/rnn2.png

Figure 8: RNN Formalization

Images/rnn5.png

Figure 9: RNN- Across time steps (1)

Images/rnn4.png

Figure 10: RNN- Across time steps (2)

- The L_i are then summed up to get the final loss L.
- The weights matrices are the same matrices but are being updated as we go along the sequence.
- We can see that $\hat{y}_t = W_{hy}^T h_t$
- The weights are shared across the sequence.
- The weights are updated using backpropagation through time.

Images/rnn3.png

Figure 11: RNN Code Sample - to easily visualize the architecture

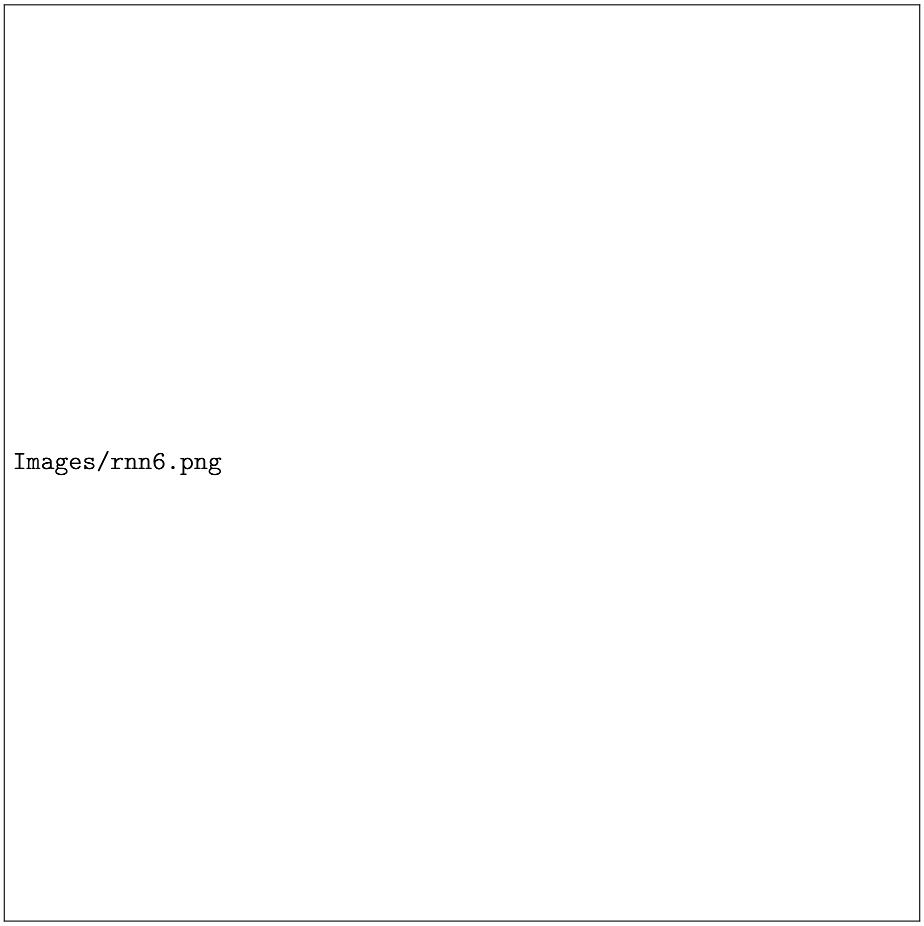

Figure 12: RNN Code from Scratch - to understand the architecture

- Instead of this you just use $tf.keras.layers.SimpleRNN(rnn\_units)$

Example of A Sequence Modeling Problem: Predict the next word

- We need to make sure of handling variable-length sequences.
- Track long-term dependencies.
- Maintain the order of the sequence.
- Share parameters across the sequence.
- Fortunately, RNNs can handle all of these requirements(some more than others).

- First Question, that we might ask ourselves is that: How do we represent the words? How should we encode language?

2.0.3 One-Hot Encoding

- Like a lookup table, so we have a vector of zeros and a 1 at the index of the word. The length of the vector is the size of the vocabulary.

2.0.4 Word Embeddings

- Learned Embedding, you might ask what dimension of the embedding we use?
- You might think at first that as the number of distinct things in my vocab increases, we should always increase the dimension of the embedding, but this is not the case, we need to find a sweet spot, because if we increase the dimension too much we lose the point of the embedding, which is to capture the semantic meaning of the words, and we might then end up with a sparse representation, which is not what we want (i.e. similar to one-Hot embedding).
- Sparse Categorical Crossentropy: This is used when your labels are integers representing class indices, rather than one-hot encoded vectors. If you have a vocabulary of size 1000, for instance, your labels will be integers from 0 to 999. $tf.keras.losses.sparse_{c}ategorical_{c}rossentropy$
- Categorical Crossentropy Loss Crossentropy: A measure of the difference between two probability distributions. In the context of classification, it's a measure of how well the predicted probability distribution (after softmax) matches the true distribution (represented by the labels).

Now How do we train the model?

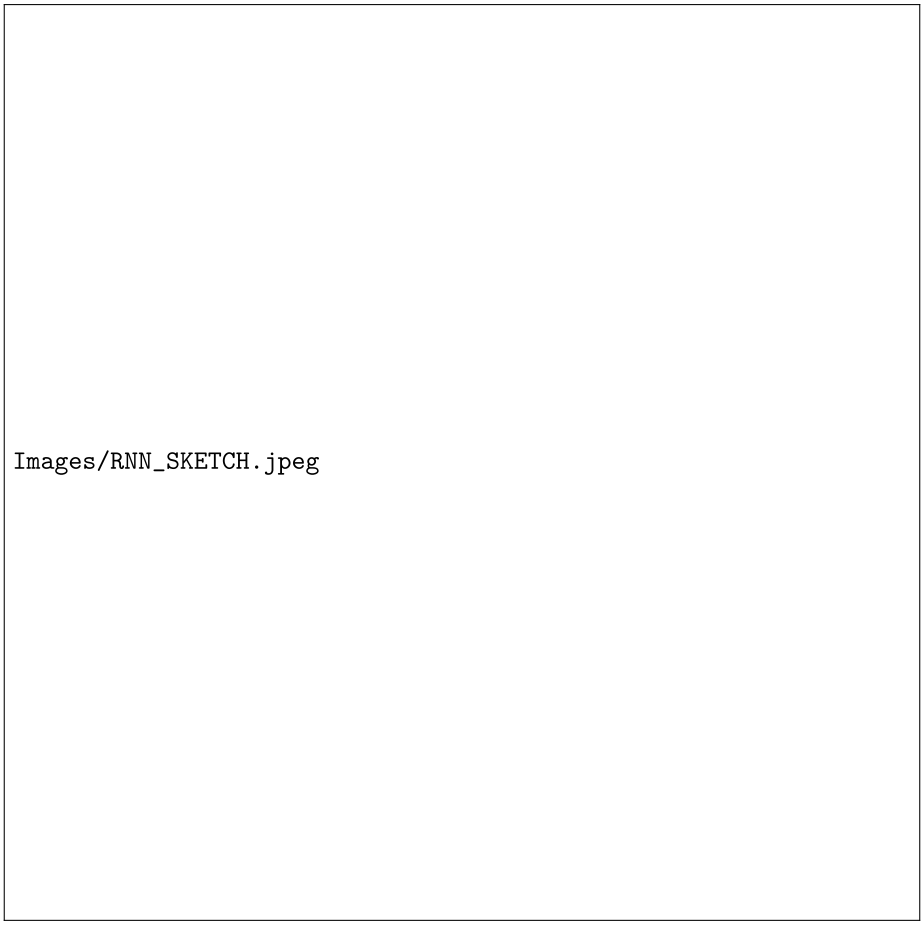

Figure 13: Training the Model

- We use Backpropagation through time (BPTT).
- But you might see that there is a problem with this, because the gradients might explode or vanish, because of the Long Sequences and Long dependencies.
- To solve this we have Many ways to do so:
- Proper use of activation functions to solve the gradient vanishing problem:

Images/2_.jpg

Figure 14: Scketch - Helper

Images/3_.jpg

Figure 15: Scketch - Helper

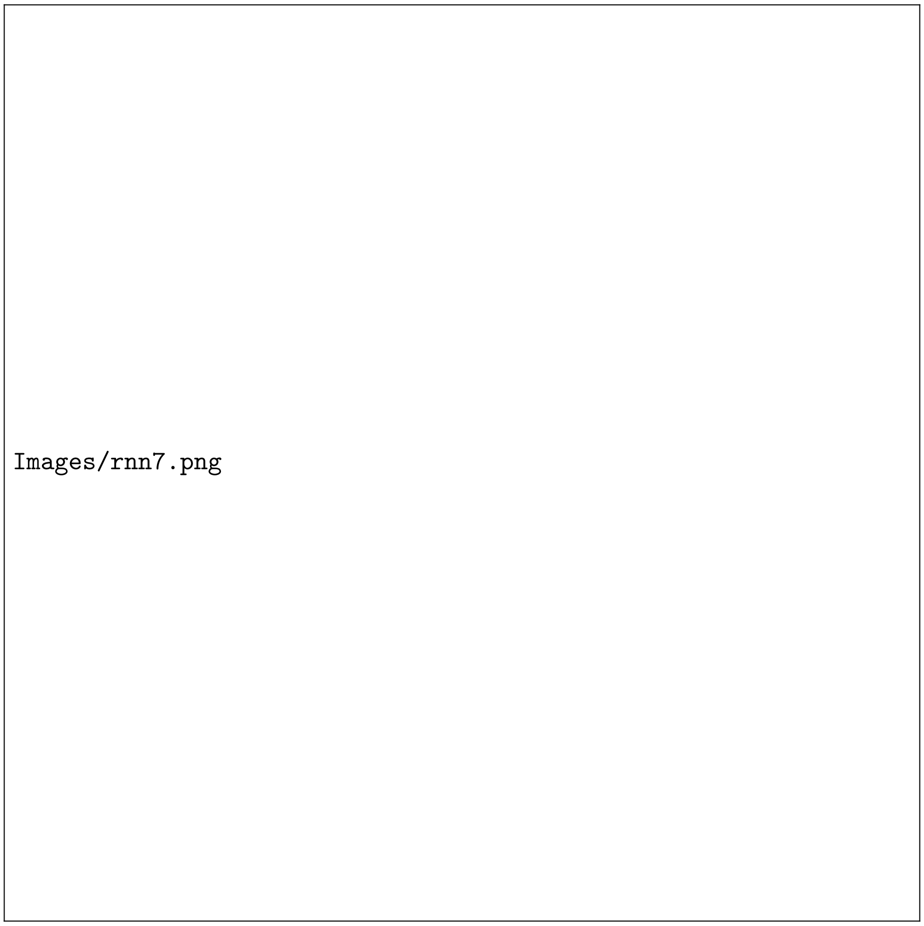

Figure 16: ReLU - A Good Choice for Activation Functions

- LSTM: Long Short Term Memory, it has a cell state that goes through the sequence, and it has gates that control the flow of information. Meaning that it can decide what to forget and what to remember.(How it decides what to forget and what to remember? It is learned from the data, so it is a learned parameter).
- Key Concepts of LSTM: Maintain the cell state, Forget gate, Input gate, Output gate.In the LSTM Backpropagation happens through time with Partially uninterrupted gradient flow.

Images/lstm.png

Figure 17: LSTM

- GRU: Gated Recurrent Unit, it is a simplified version of the LSTM, it has a reset gate and an update gate. It is faster to train than the LSTM.

Images/gru.png

Figure 18: GRU

- We can see that ultimately we can see them as a black box that takes an input and gives an output, but each model whether (Basic RNN, LSTM, GRU) has its own architecture. Each model chooses what to pass to the next time step. RNN: Passes everything(you might think that this is a good thing, meaning this way we are gaining all information but that is not the case, because of the gradient vanishing problem), LSTM and GRU: Chooses what to pass based on the gates.

- So we can see that even with LSTM and GRU we still have the problem of the vanishing gradient, but it is less severe than the RNN. So what is the solution?

Transformers and Attention Mechanism

- Intuition is to build a neural network that will choose what is an important part of the sequence and what is not, so it will focus on the important parts of the sequence.
- What we will focus on is self-attention, so we will focus on the important parts of the sequence
- Intuition behind Self-Attention: Identify which parts to attend to, Step 2 Extract the features with high attention
- So in self-attention, we first take the input as a whole (removing the recursion). But we need to make sure we still keep information about the order, how do we do this? We use positional encoding.

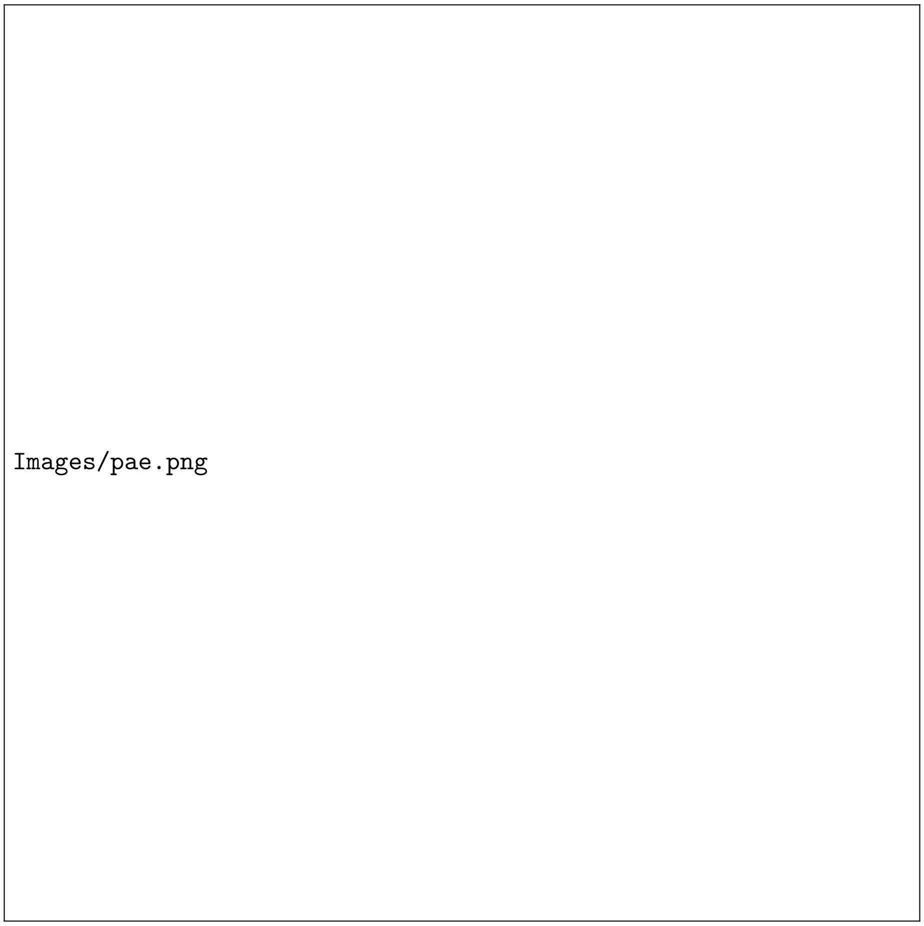

Figure 19: Position-aware Embeddings

Images/qkv.png

Figure 20: Query, Key, Value

- Notice that we took the input all at once, we then used our regular word-embedding, then we use position information to keep the order of the sequence to finally get the Position-aware Encoding. So in this way our final embeddings contain information about the order.
- Now we need a way to extract query, key, and value from the input.
- So what we do is that we take the PAE and we triplicate it, and then feed each one of them to a separate Linear neural network to get the Query, Key, and Value.
- Now how do we compute the attention score? We need a way to compute the attention weighting for each word in the sequence.
- To do so we need to compute the pairwise similarity between the query

and the key. To do so we use the dot product (also known as cosine similarity).

Images/attention_score.png

Figure 21: Attention Score

- Now we have how similar each word is to the query, but we need to normalize this, so we use the softmax function.

Images/attention_score2.png

Figure 22: Attention Score - Normalized

- Remember that our goal is to extract the most important features in the input based on the highest attention score.

Images/attention_score3.png

Figure 23: Attention Score - Extracting the most important features

- Now let's put it all together:

Images/all_.png

Figure 24: Single Self-Attention Head

- How Self-Attention Works:
  Formulate Queries:
  For each word, create a query vector that represents what that word wants to know from the rest of the sequence.
  Match Queries to Keys:
  Compare the query with the keys of all words using the dot product. This comparison gives a similarity score for each word.
  Assign Attention Weights:
  Use the similarity scores to assign weights to each word's value. Higher similarity results in higher attention weights.
  Compute the Output:
  Combine the values using the attention weights to form a context-aware output for the current word.
  Importance of Q, K, and V:
  Query: Determines the focus or what information is needed.
  Key: Helps in matching relevant information in the sequence.
  Value: Provides the actual content that is used to generate the final output.

- So by computing $Q.K^T$ we are essentially getting a k x k matrix where each element is the similarity between the query and the key. Then we use softmax which will keep the k x k matrix the same size but will normalize it. Then we multiply this by the V which is the value matrix to get the final output.
- Now the last question: How does this fit in the transformer architecture?
- Each attention head computes its own set of query (Q), key (K), and value (V) vectors by applying different linear transformations to the input embeddings.
- The attention head then calculates attention scores by comparing the query vector to all key vectors in the sequence, using the dot product. These scores are used to weight the value vectors.
- The attention head combines the weighted value vectors to produce an output. This output emphasizes the parts of the input that are most relevant according to the head's specific focus.
- In models like the Transformer, multiple self-attention heads are used in parallel. The outputs from all heads are combined (concatenated) and then passed through another linear layer to produce the final output of the self-attention mechanism.
- So key points to remember:

- Attention is the foundation of the transformer architecture, enabling the model to focus on different parts of the input sequence.
- Self-Attention Head: A single pathway that computes attention for a specific aspect of the input.
- Multi-Head Attention: Combines multiple self-attention heads to capture various relationships and dependencies in the data.
- Diversity of Focus: Each head in multi-head attention can focus on different parts or features of the input, providing a more comprehensive understanding.

Images/multi_head.png

Figure 25: Multi Head Attention

- We can see how in the first row we have the output of the softmax function, and then we multiply this by the value matrix to get the final output.

(Which is what we need to focus on)

- We can see that each attention Head can be trained on focusing on a particular aspect of the input.
- Once we combine these self-attention heads we get the transformer architecture.