# QUESTION-ANSWERING ON SQUADv1.1 DATASET

Implementation of a neural architecture for automated question answering called R-NET, which is developed by the Natural Language Computing Group of Microsoft Research Asia.
This post describes the model and the challenges I faced while implementing it.
[GitHub]

---

## PROBLEM STATEMENT

Given a passage and a question, the task is to predict an answer to the question based on the information found in the passage. The **SQuAD dataset** further constrains the **answer to be a continuous sub-span** of the provided passage. Answers usually include non-entities and can be long phrases. The neural network needs to "understand" both the passage and the question in order to be able to give a valid answer. Here is an example from the dataset.
EX::
**Passage:** Tesla later approached Morgan to ask for more funds to build a more powerful transmitter. When asked where all the money had gone, Tesla responded by saying that he was affected by the Panic of 1901, which he (Morgan) had caused. Morgan was shocked by the reminder of his part in the stock market crash and by Tesla's breach of contract by asking for more funds. Tesla wrote another plea to Morgan, but it was also fruitless. Morgan still owed Tesla money on the original agreement, and Tesla had been facing foreclosure even before construction of the tower began.
**Question:** On what did Tesla blame for the loss of the initial money? Answer: Panic of 1901

**Key Features of SQuAD**:

- It is a closed dataset meaning that the answer to a question is always a part of the context and also a continuous span of context.

- So the problem of finding an answer can be simplified as finding the start index and the end index of the context that corresponds to the answers.

- 75% of answers are less than equal to 4 words long.

---

## **ARCHITECTURE OF R-NET** [ARCHITECTURE]

The process consists of several steps:

1. Encode the question and the passage.
2. Obtain question-aware representation for the passage.
3. Apply self-matching attention on the passage to get its final representation.
4. Predict the interval which contains the answer of the question using the Pointer Network.

Each of these steps is implemented as some sort of recurrent neural network. The model is trained end-to-end.
I am using GRU cells for all RNNs rather than LSTMs, as reported by the authors, performance is similar to that of LSTMs but they are computationally cheaper.

Here is a chart of an original GRU cell along with the equation -

$$
\begin{aligned}
z_t &= \sigma(W_z . [h_{t-1}, x_t) \\
r_t &= \sigma(W_r . [h_{t-1}, x_t) \\
\hat{h}_t &= tanh(W . [r_t \circ h_{t-1}, x_t) \\
h_t &= (1 - z_t) \circ h_{t-1} + z_t \circ \hat{h}_t
\end{aligned}
$$

Some parts of R-NET architecture require to use tensors that are neither part of a GRU state nor part of input at a time. These are **“global” variables** that are used in all timesteps. These global variables are called ***non-sequences***.
To make it easier to create GRU cells with additional features and operations we’ve created a utility class called **WrappedGRU** which is a **base class for all GRU modules**. **WrappedGRU** supports operations with non-sequences and sharing weights between modules. Keras doesn’t directly support weight sharing, but instead, it supports layer sharing and we use SharedWeight layer to solve this problem (SharedWeight is a layer that has no inputs and returns tensor of weights).
WrappedGRU supports taking SharedWeight as an input.

## **Encoding the question and passage -**

- Consists of **text encoding**.

- First before encoding the text, **preprocessing** is carried out where the data is represented by splitting it into tokens and then converting all the tokens to corresponding vectors using gensim and GloVe word embeddings.

- Each word is represented by a **concatenation of two vectors**: its **GloVe vector** and **another vector that holds character level information**. To obtain character level embeddings we use an Embedding layer followed by a Bidirectional GRU cell wrapped inside a TimeDistributed layer. Basically, each character is embedded in ***H*** dimensional space, and a BiGRU runs over those embeddings to produce a vector for the word. The process is repeated for all the words using TimeDistributed layer.
  [CODE on GITHUB]

```
TimeDistributed(Sequential([
    InputLayer(input_shape=(C,), dtype='int32'),
    Embedding(input_dim=127, output_dim=H,   mask_zero=True),
    Bidirectional(GRU(units=H))
]))
```

- When the word is missing from GloVe, we set its word vector to all zeros.

  The network takes the preprocessed question and the passage applies masking on each one and then encodes them with 3 consecutive bidirectional GRU layers.

```
# Encode the passage P
uP = Masking() (P)
for i in range(3):
    uP = Bidirectional(GRU(units=H,
                           return_sequences=True,
                           dropout=dropout_rate)) (uP)
uP = Dropout(rate=dropout_rate, name='uP') (uP)

# Encode the question Q
uQ = Masking() (Q)
for i in range(3):
```

```
    uQ = Bidirectional(GRU(units=H,
                           return_sequences=True,
                           dropout=dropout_rate)) (uQ)
uQ = Dropout(rate=dropout_rate, name='uQ') (uQ)
```

- After encoding the passage and the question we finally have their vector representations $u^P$ and $u^Q$ respectively.

## Question-Aware Passage Representation -

- **QuestionAttnGRU** is a complex extension of a recurrent layer (extends **WrappedGRU** and overrides the step method by adding additional operations before passing the input to the GRU cell), which is responsible for obtaining question-aware passage representation.
  [CODE on GITHUB]

```
vP = QuestionAttnGRU(units=H,
             return_sequences=True) ([
                 uP, uQ,
                 WQ_u, WP_v, WP_u, v, W_g1
             ])
```

- The vectors of question aware representation of the passage are denoted by $v^P$. $u_t^P$ is the vector representation of the word $t$ of passage $P$, $u^Q$ is the matrix representation of the question(each row corresponds to a single word).

- In QuestionAttnGRU first we combine three things:
  - The previous state of the GRU ( $v_{t-1}^P$ )
  - Matrix representation of the question ( $u^Q$ )
  - Vector representation of the passage ( $u_t^P$ ) at the $t^{th}$ word.

- The dot product of each input is computed with the corresponding weights, then sum-up all together after broadcasting them into the same shape.

The outputs of dot( $u_t^P$ , $W_u^P$ ) and dot( $v_{t-1}^P$ , $W_v^P$ ) are vectors, while the output of dot( $u^Q$ , $W_u^Q$ ) is a matrix, therefore we broadcast (repeat several times) the vectors to match the shape of the matrix and then compute the sum of three matrices.

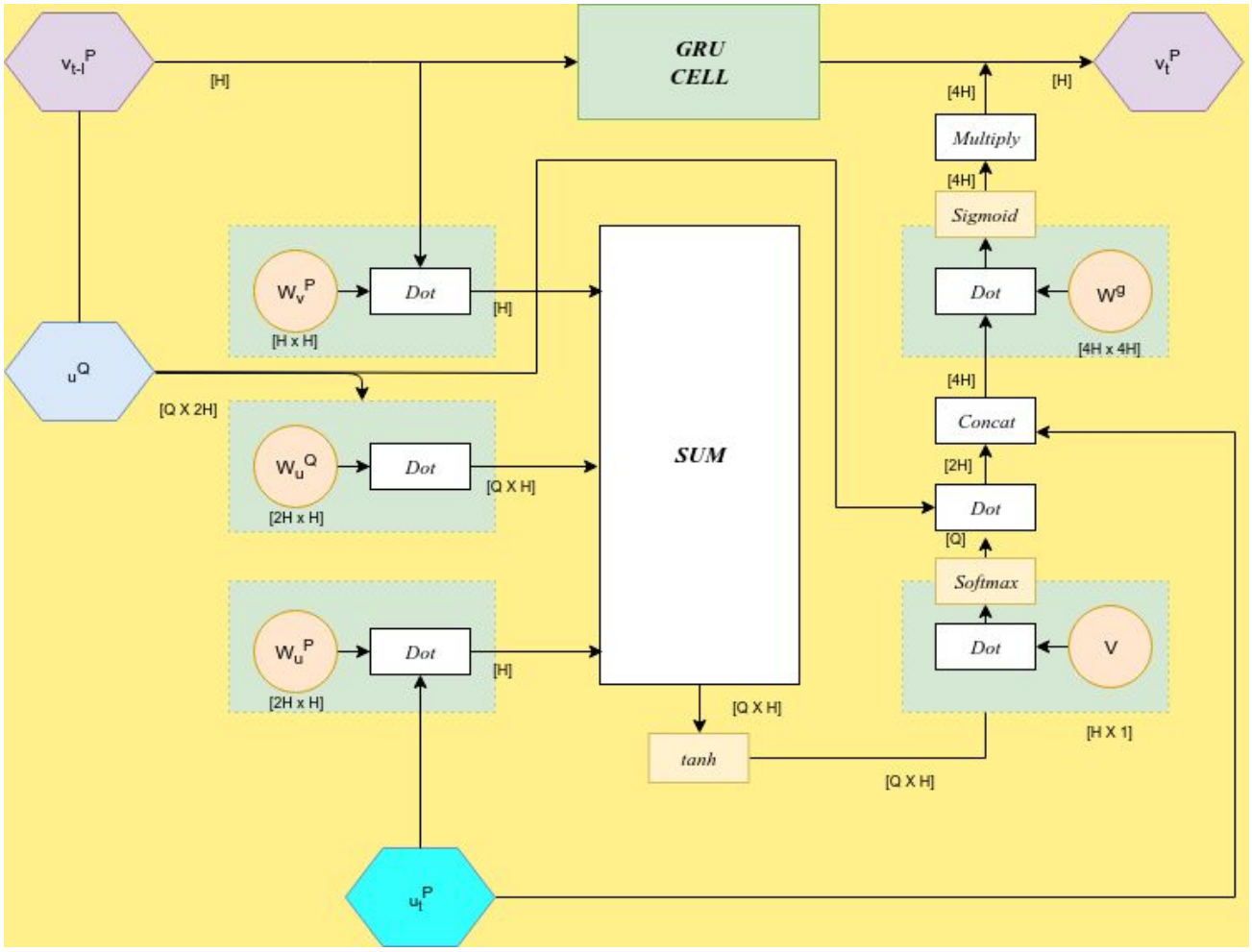
Then, apply $tanh$ activation on the result.

The output of this operation is then multiplied (dot product) by a weight vector $V$ , after which $softmax$ activation is applied.

The output of the $softmax$ is a vector of non-negative numbers that represent the **"importance"** of each word in the question. This kind of vectors are often called `attention vectors`.

When computing the dot product of $u^Q$ (matrix representation of the question) and the attention vector, we obtain a single vector for the entire question which is a weighted average of question word vectors (weighted by the attention scores). The intuition behind this part is that we get a representation of the parts of the question that is relevant to the current word of the passage.

This representation, denoted by $c_t$, depends on the current word, the whole question and the previous state of the recurrent cell.




[drawn using draw.io]

- The authors of R-NET did one more step. They applied an additional gate to the concatenated vector $[c_t, u_t^P]$. The gate is simply a dot product of some new weight matrix $W_g$ and the concatenated vector, passed through a sigmoid activation function. The output of the gate is a vector of non-negative numbers, which is then (element-wise) multiplied by the original concatenated vector. The result of this multiplication is finally passed to the GRU cell as an input.

## Self-Matching Attention to get Passage final representation -

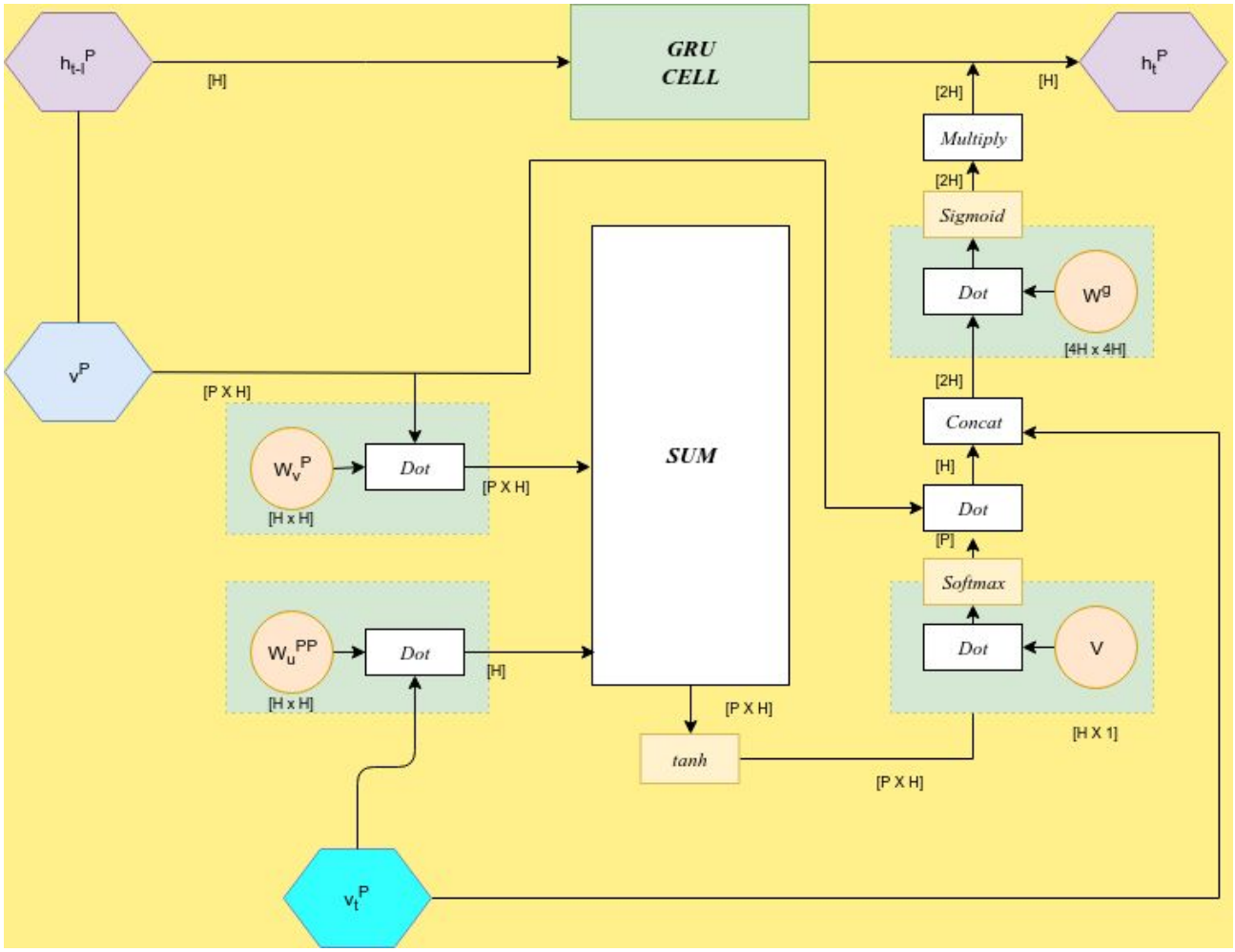
- [CODE on GITHUB]

```
hP = Bidirectional(SelfAttnGRU(units=H,
                               return_sequences=True)) ([
```

```
                        vP, vP,
                        WP_v, WPP_v, v, W_g2
                    ])
hP = Dropout(rate=dropout_rate, name='hP') (hP)
```

- The output of the previous step (Question attention) is denoted by $v^P$. It represents the encoding of the passage while taking into account the question. $v^P$ is passed as an input to the **self-matching attention** module.
  Recurrent Networks can only memorize limited passage context in practice despite its theoretical capability.
  Self-matching attention module attempts to augment the passage vectors by information from other relevant parts of the passage.

- The output of the self-matching GRU cell at time is denoted by $h_t^P$.



[drawn using draw.io]

- The dot products of weights $W_u^P$ is computed with the current word vector $v_t^P$, and $W_v^P$ with the entire matrix $v^P$, then add them up and apply $tanh$ activation. Next, the result is multiplied by a weight-vector $V$ and passed through $softmax$ activation, which produces an attention vector.
  The dot product of the attention vector and $v^P$ matrix, again denoted by $c_t$, is the weighted average of all word vectors of the passage that are relevant to the current word $v_t^P$.
  $c_t$ is then concatenated with $v_t^P$ itself. The concatenated vector is passed through a gate and is given to GRU cell as an input.

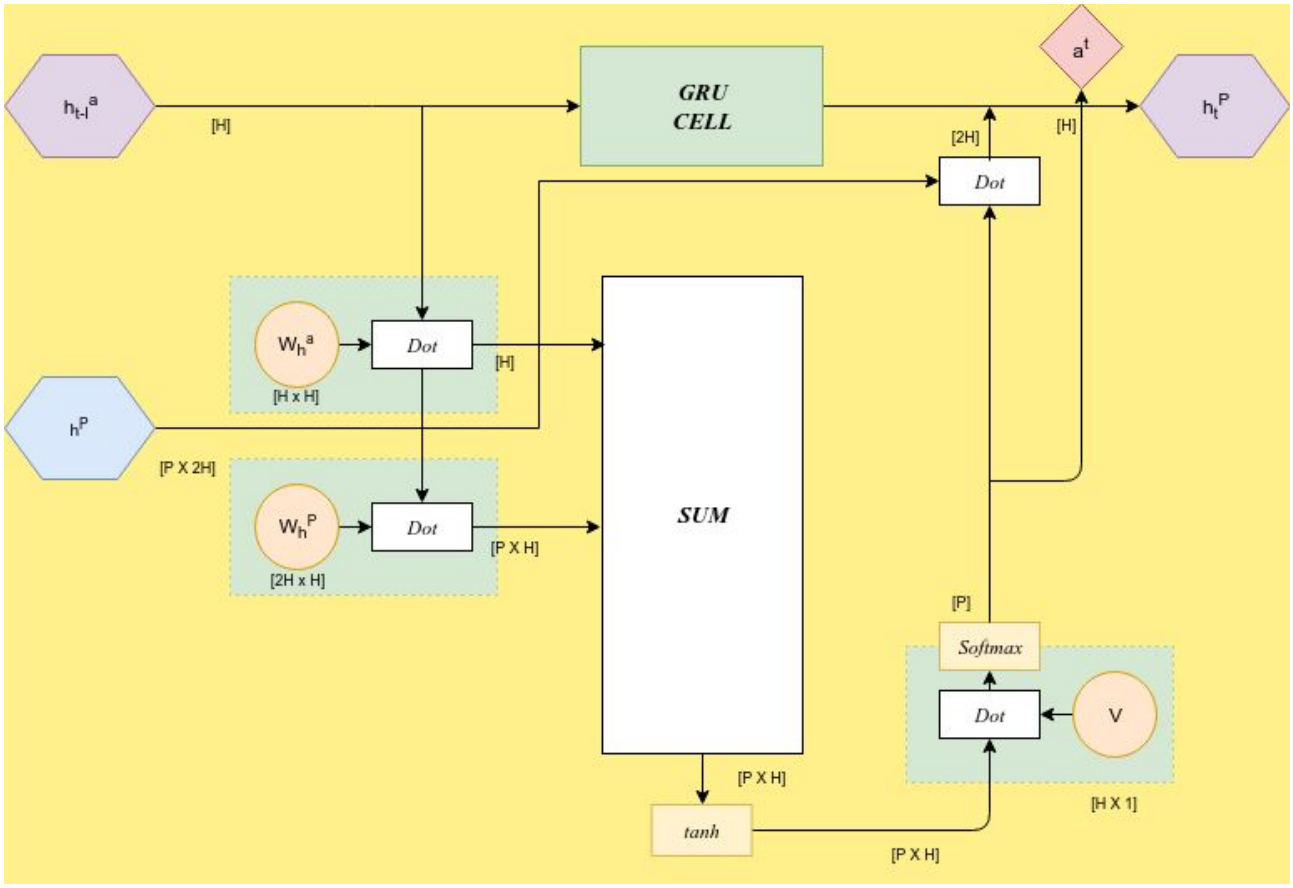
## PointerNetwork to predict span/interval of answer -

- [CODE on GITHUB]

```
rQ = QuestionPooling() ([uQ, WQ_u, WQ_v, v])
rQ = Dropout(rate=dropout_rate, name='rQ') (rQ)

...

ps = PointerGRU(units=2 * H,
                return_sequences=True,
                initial_state_provided=True,
                name='ps') ([
            fake_input, hP,
            WP_h, Wa_h, v, rQ
        ])

answer_start = Slice(0, name='answer_start ') (ps)
answer_end = Slice(1, name='answer_end') (ps)
```

- QuestionPooling is the **attention pooling** of the whole question vector $u^Q$. Its purpose is to create the **first hidden state** of **PointerGRU**.
  $h^P$ is the output of the previous module and it contains the final representation

of the passage. It is passed to this module as an input to obtain the final answer.




[drawn using draw.io]

- PointerGRU is a recurrent network that works for just two steps. The **first step** predicts the **first word** of the answer span, and the **second step** predicts the **last word**.
  Both $h^P$ and the previous state of the PointerGRU cell are multiplied by their corresponding weights $W$ and $W_v^a$. The **initial hidden state** of the PointerGRU is the output of **QuestionPooling**.
  The products are then summed up and passed through $tanh$ activation. The result is multiplied by the weight vector $V$ and $softmax$ activation is applied which outputs scores over $h^P$.
  These scores, denoted by $a^t$ are probabilities over the words of the passage. Argmax of $a^1$ vector is the predicted starting point, and argmax of $a^2$ is the predicted final point of the answer.
  The **hidden state** of **PointerGRU** is determined based on the dot product of $h^P$ and $a^t$, which is passed as an input to a simple GRU cell.

So, unlike all previous modules of R-NET, the **output of PointerGRU** (the red diamond at the top-right corner of the chart) is different from its hidden state.

---

## IMPLEMENTATION DETAILS

- **Layers with Masking Support -**
  One of the most important challenges in training recurrent networks is to handle different lengths of data points in a single batch. Keras has a **Masking layer** that handles the basic cases. We use it while encoding the passage and question in the encoding layer. But **R-NET** has more complex scenarios. For example, in all **attention pooling modules** used which is applied along "time" axis (e.g. over the words of the passage), we don't want to have positive probabilities after the last word of the sentence. So implemented a custom Softmax function which supports masking.

```
def softmax(x, axis, mask):
    m = K.max(x, axis=axis, keepdims=True)
    e = K.exp(x - m) * mask
    s = K.sum(e, axis=axis, keepdims=True)
    s = K.clip(s, K.floatx(), None)
    return e / s
```

  $m$ is used for numerical stability. To support masking we multiply $e$ by the mask. We also clip $s$ by a very small number, because in theory, it is possible that all positive values of $e$ are outside the mask.

- **Slice Layer -**
  It is supposed to slice and return the input tensor at the given indices. It also supports masking. The slice layer in **R-NET** model is needed to extract the final answer (i.e. the `interval_start` and `interval_end` numbers).
  The final output of the model is a tensor with the shape `(batch x 2 x passage_length)`. The first row contains probabilities for `answer_start` and the second one for `answer_end`, that's why we need to slice the rows first and then extract the required information.

- **Generators -**
  [CODE on GITHUB]
  Keras supports batch generators which are responsible for generating one batch per each iteration. One benefit of this approach is that the generator is working on a separate thread and is not waiting for the network to finish its training on the previous batch.

- **Bidirectional GRUs -**
  R-NET uses multiple bidirectional GRUs. The common way of implementing BiRNN is to take two copies of the same network (without sharing the weights) and then concatenate the hidden states to produce the output. One can take the sum of the vectors instead of concatenating them, but concatenation seems to be more popular (that's the default version of Bidirectional layer in Keras).

- **Weight Sharing -**
  Weights were shared among :
  - matrix between QuestionAttnGRU and QuestionPooling layers,
  - matrix between QuestionAttnGRU and SelfAttnGRU layers,
  - vector between all four instances (it is used right before applying softmax).

  We didn't share the weights of the **"attention gates"**. The reason is that we have a mix of uni- and bidirectional GRUs that use this gate and require different dimensions.

- **Hyperparameters -**
  The authors tell many details about hyperparameters. Hidden vector lengths are **75** for all layers. As we concatenate the hidden states of two GRUs in bidirectional, we effectively get 150-dimensional vectors.
  AdaDelta optimizer is used to train the network with `learning rate=1`, and . Nothing is written about the size of batches, or the way batches are sampled. We used `batch_size=100`.
  We couldn't get good performance with 75 hidden units. The models was quickly overfitting. We got our best results using 45 dimensional hidden states.

- **Weight Initialization -**
  The report doesn't discuss weight initialization. So, default initialization schemes of Keras was used. In particular, Keras uses orthogonal initialization for recurrent

connections of GRU, and uniform (Glorot, Bengio, 2010) initialization for the connections that come from the inputs. Glorot initialization was also used for all shared weights.

---

## RESULTS

SQuAD dataset uses two performance metrics: **exact match (EM)** and **F1-score (F1)**. Human performance is estimated to be **EM=82.3% and F1=91.2%** on the test set.

R-NET was till August 2017 the best model on Stanford QA benchmark among single models with **EM=82.136% and F1=88.126%** on the test set.

The best performance we got so far with our implementation is
**EM=57.52% and F1=67.42%** on the development set.
One of the reason was that model was not trained enough due to lack of resources and time. However, it would still be impossible to reach same level of metric score as mentioned in the report/paper.

The results we got would put R-NET at the bottom of the SQuAD leaderboard.
The model is available on GitHub.
The R-NET's technical report is pretty good in terms of the reported details of the architecture compared to many other papers. Probably misunderstood several important details or have bugs in the code. Any feedback will be appreciated.

---

## FURTHER IMPROVEMENTS

- **Embeddings :** Can use other embeddings than GloVe like fasttext embeddings, ELMo, etc.

- **Summarization :** Can use extractive summarization to obtain important parts of passage and can also encode it and add to the passage representation. This can

help to further improve the question-aware passage representation.

- **Preprocessing :** Some preprocessing techniques can be applied to further improve the results. Different tokenizers can be also applied like spacy tokenizer, CoreNLP tokenizer, regex tokenizer, etc but will not tend to improve the result by any significant amount.

- **Architecture :** Different types of Attention module can be applied like BiDAF Attention, Dynamic Coattention, etc.

---

DISCLAIMER :: Still, there is a lot of scope of improvement and the thing that limits me is my knowledge which I am hoping to improve bit-by-bit, day-by-day.