Knowledge Graph Convolutional Networks

Why haven't machines taken over the world yet?

It seems quite clear that machines should be able to outperform humans in many more tasks than they currently can, or at least that they should be able to make truly smart predictions. I'm sure we can all relate to a moment when an app made us a recommendation that didn't make any logical sense. You only have to look as far as the recommender system that recommended the product you only just bought, or the spam filter that stole a reply from someone you messaged. In practice, we find that we can't trust machines with decision-making on our behalf.

Intuitively, it feels like our machines are missing something, some capability, such that if we aren't very careful about how we teach them then they'll miss the blindingly obvious. Why is this?

If our machines are missing key elements, then there must be unexplored territory that we can leverage to improve the performance of our models. We can start by examining the abilities that we have ourselves.

There are a few key skills that humans possess that are clearly necessary for making good decisions: **learning [1]** and **logical reasoning [2]** on top of a pre-existing **knowledge base (human memory) [3]**.

Thus far in mainstream 'AI' of these three, we've only really cracked learning.

Skill 1 — Learning

The underlying, and well-recognised, pitfall of doing learning alone is that the learning is shallow; the result is always a complicated black-box mapping from a flat set of symptoms to a conclusion. Learning machines aren't encoding an understanding of the deeper causes of the outcomes they see in their training data. This means that learning on its own is inclined to miss the point.

We ask: is there a way to give our machines the other skills that we possess, that they lack? Armed with those skills, they can make decisions that we can trust.

Skill 2 — Being Logical

Deduction or *reasoning* is second-nature to humans. We use a mental model of rules to predict what will happen in our surroundings. Take this example:

My dinner is in the pan, the pan is on the stove, and the food is simmering. This implies that the pan is hot. Given that the pan is hot, if I touch the pan it'll burn my hand. Therefore the pan is dangerous.

This line of reasoning is obvious to us (except for those odd occasions where you forget and you do burn yourself). In fact, this kind of deduction is so fundamental to our lives that it's hard to spot when we're using it.

Reasoning is central to our existence, and to human intelligence. Consequently, it's useful to build tools that can automate that thinking for us at great scale and complexity. That's just the start, there's even more to be gained from it.

Get More from Less

Deduction takes place over a certain set of facts, and concludes with the generation of new facts. This means that given information and the rules that govern a domain, exponentially more information can be derived.

This sounds particularly enticing in the world of machine learning, where the size of dataset is often the performance bottleneck. Reasoning promises to let additional knowledge be derived from the existing. This radically augments the value of the data already available.

The trouble is, as it stands, the vast majority of machine learning models cannot perform any kind of deduction.

Ideally, we want our learning systems to make use of the rules that govern a domain, and correlate the outcomes of those rules. In this way, they can gain an *understanding* of the dynamics of the field.

Skill 3— Storing Knowledge

Reasoning relies upon the context of data. This is necessary in order to automatically know in which particular circumstances a new fact can be inferred. If we created a formula for this, we would say:

Data + Context = Knowledge

What do we really mean by context?

Context in this case means the information governing the structure of data. To use reasoning for a given datapoint we need to know what type of data it is, and how it is inter-related to other datapoints.

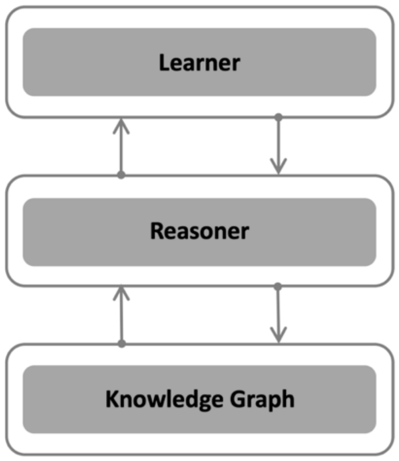
The information of the real world isn't made up of a set of column names that we can put into a table. Tabulation is a method we've been using for many years to de-complicate the problem domain. Unfortunately, this process removes the important context: the inter-relations between data-points. It makes sense to throw away structural information if you don't have a technique that can use it. But, if you can use it, then you can build a much more intelligent solution.

We can consider how we store knowledge in our own heads. People don't put their memories in an ordered system of filing cabinets in the back of their heads (desirable as that might seem). Instead, we hold a web of connected experience: a **knowledge base**. For our intelligent systems that component is called a **knowledge graph**. This technology enables structured, contextual data storage and reasoning across its knowledge content.

Skills 1+2+3 = Learning using Reasoned Knowledge

We want to combine all three skills together. The natural progression is to shift from learning over *flat, non-contextual data* to learning over *reasoned, contextual knowledge*. There are numerous benefits that we can derive from this:

- Learning **based on facts inferred** via reasoning
- **Implicitly embed the context** of each datapoint into the learned model for better decision-making
- Prediction that may **generalise beyond the scope of the training data** according to the logical rules of the domain
- A **transparent and queryable** graph of the underlying domain knowledge
- **Reduced quantity of training data required**



Conceptual architecture for an Intelligent System

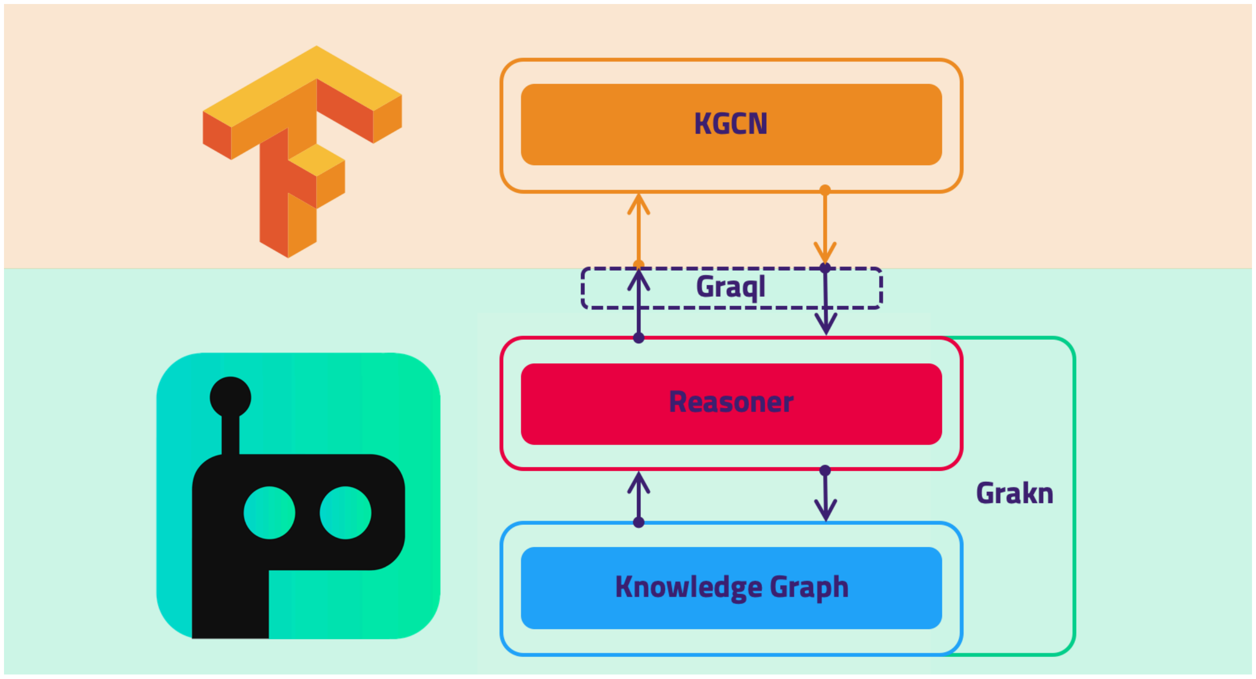
How do we do it?

To learn based on reasoned knowledge we need to learn over a knowledge graph. Graph learning is a new research area, where some of the most promising models are Graph Convolutional Networks (GCN). Knowledge graph learning research is in its absolute infancy.

In general, we need an architecture as depicted: a learner asks questions of a knowledge graph via logical reasoning. In this way, any inference of facts made by the reasoner can be performed as and when the learner demands information from the knowledge graph. As a result, the knowledge graph isn't required to store all of the possible facts that the reasoner might infer, they can be generated at query-time.

Knowledge Graph Convolutional Networks (KGCNs)

We're in a good starting position given that Grakn is a knowledge graph platform with automated reasoning out-of-the-box. The next step is to build a learner on top of this backend in a way that exploits reasoning. We exploit reasoning for free if our learning framework makes use of *Graql*, Grakn's query language. Every time the learner queries via Graql it receives both facts explicitly stored and facts implicitly derived.

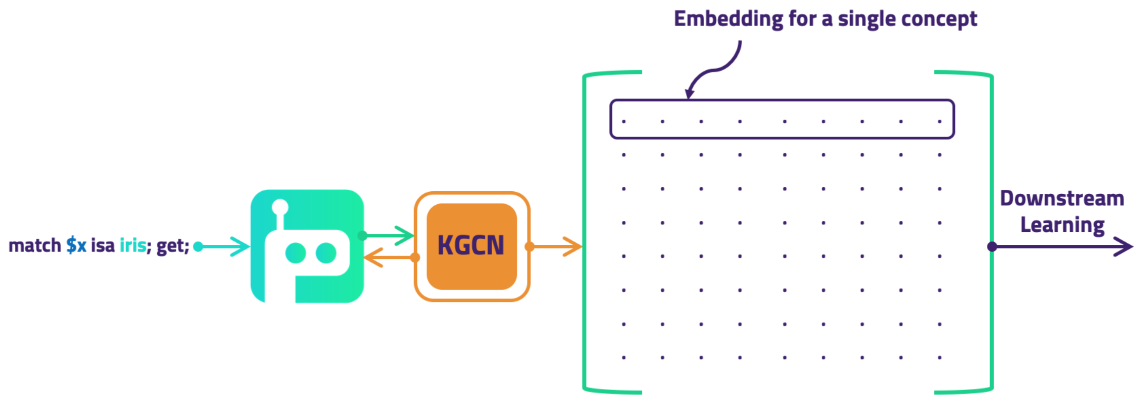


Top-level architecture for a KGCN built on Grakn

We are pleased to present *Knowledge Graph Convolutional Networks*. This is a novel variant of graph convolution, building upon the latest research. KGCNs make use of all of the available information a knowledge graph holds: node types, edge types, node features and structure. As they do so, they utilise logical reasoning to also find inferred information of these types.

The KGCN project is part of an open-source Python library, making extensive use of TensorFlow. KGCNs provide Grakn users with a method of building a vector representation for any concept held in their knowledge graph. A concept's vector representation should encapsulate its part in the graph both locally and globally. In this way, it can usefully be passed onwards to a domain-specific learning pipeline.

This makes learning pipelines much simpler. Start by querying for the concepts to use as the training/validation/test/prediction knowledge-sets using Graql. These queries can be as complex as necessary. Grakn will return the matching concepts, which can be passed to the KGCN in order to build an embedding for each of them (pictured).



From a query, to concepts, to concept embeddings

Acquiring these embeddings is the most robust interface to the tools available to the community. It is essentially a method of flattening the meaning of any concept into an array. That array is then directly ingestible by common Python machine learning tools (Numpy, SciPy, Keras, TensorFlow, PyTorch, Theano...).

KGCNs provide the fundamental bridge from knowledge stored in Grakn

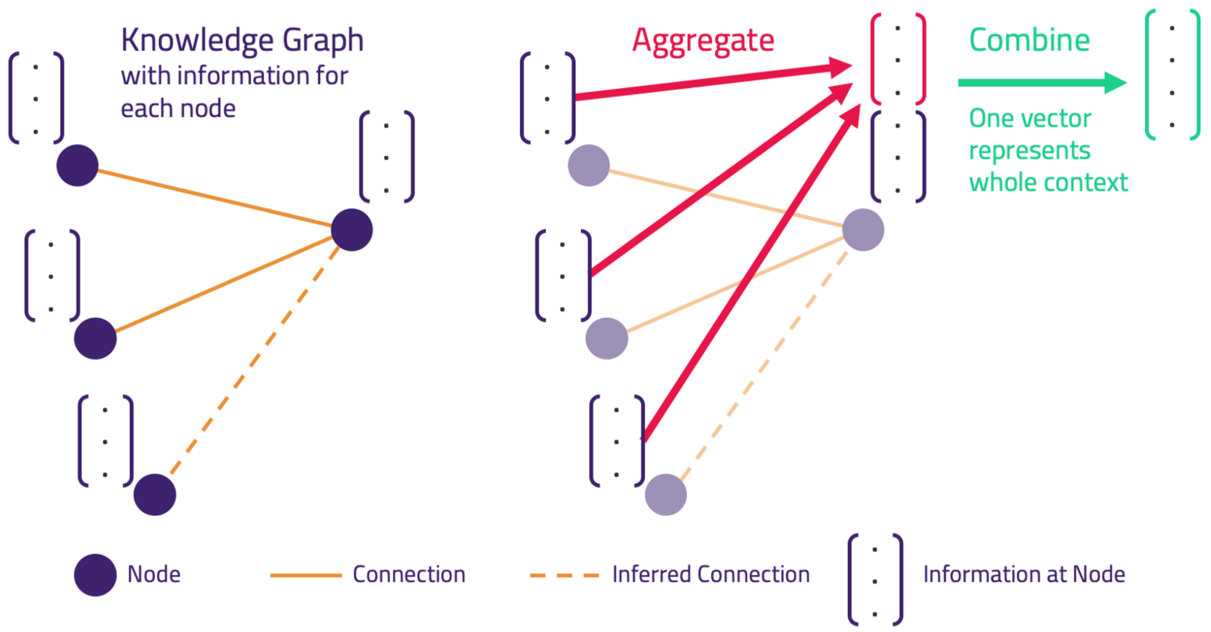
to the input of your own learning pipeline.

Methodology

The principles of the implementation are based on GraphSAGE, from the Stanford SNAP group, heavily adapted to work over a knowledge graph. Now we introduce the key components and how they interact.

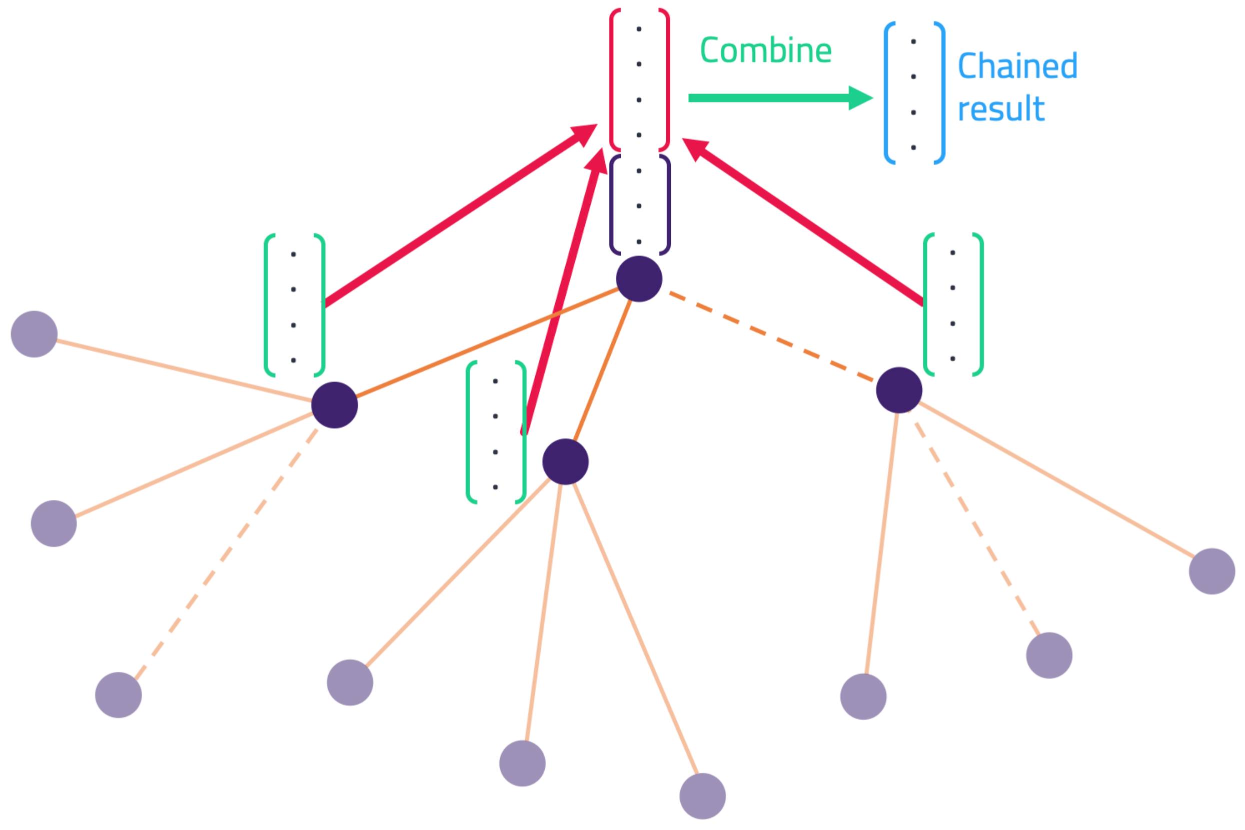
KGCN

A KGCN is responsible for deriving embeddings for a set of Things (and thereby directly learn to classify them). We start by querying Grakn to find a set of labelled examples. Following that, we gather data about the context of each example Thing. We do this by considering their neighbours, and their neighbours' neighbours, recursively, up to *K* hops away.



We retrieve the data concerning this neighbourhood from Grakn (diagram above). This information includes the *type hierarchy*, *roles*, and *attribute value* of each neighbouring Thing encountered, and any inferred neighbours (represented above by dotted lines). This data is compiled into arrays to be ingested by a neural network.

Via operations Aggregate and Combine, a single vector representation is built for a Thing. This process can be chained recursively over *K* hops of neighbouring Things. This builds a representation for a Thing of interest that contains information extracted from a wide context.



In supervised learning these embeddings are directly optimised to perform the task at hand. For multi-class classification this is achieved by passing the embeddings to a single subsequent dense layer and determining loss via softmax cross entropy (against the example Things' labels); then, optimising to minimise that loss.

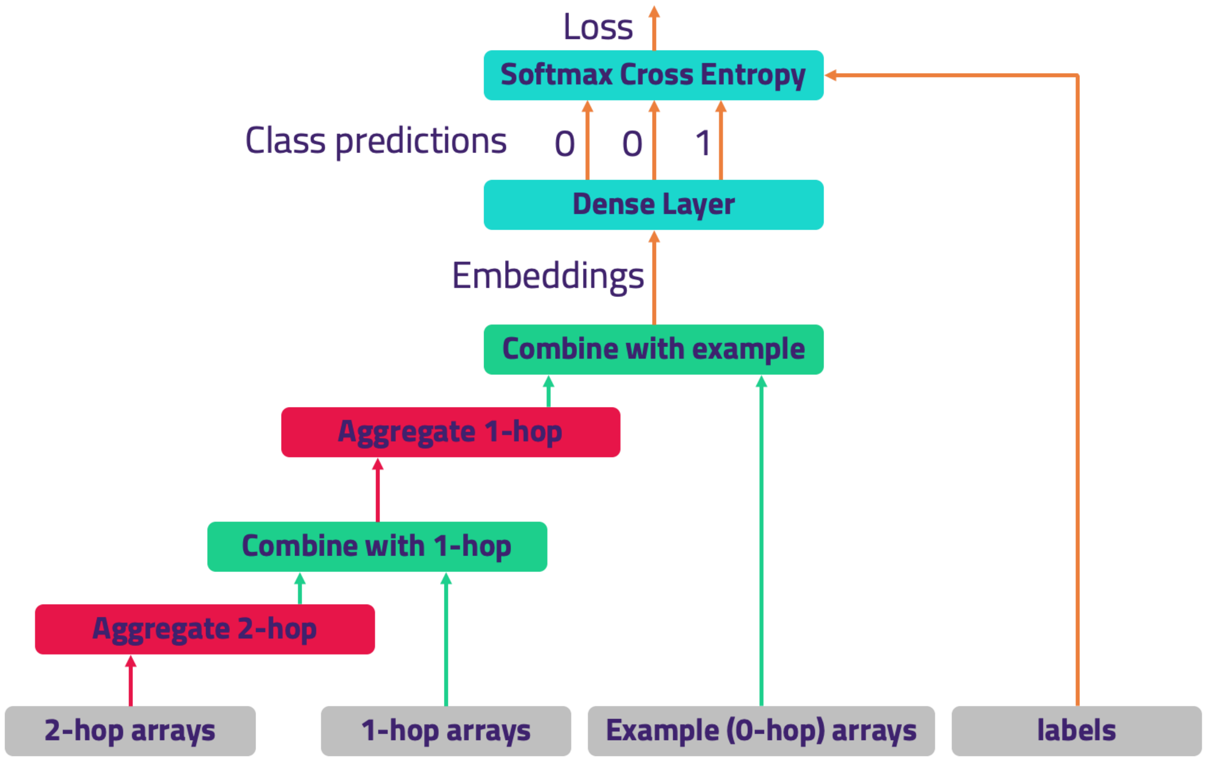
A KGCN object brings together a number of sub-components, a Context Builder, Neighbour Finder, Encoder, and an Embedder.

The input pipeline components are less interesting, so we'll skip to the fun stuff. You can read about the rest in the KGCN readme.

Embedder

To create embeddings, we build a network in TensorFlow that successively aggregates and combines features from the K hops until a 'summary' representation remains — an embedding (diagram below).

To create the pipeline, the Embedder chains Aggregate and Combine operations for the K-hops of neighbours considered. e.g. for the 2-hop case this means Aggregate-Combine-Aggregate-Combine.

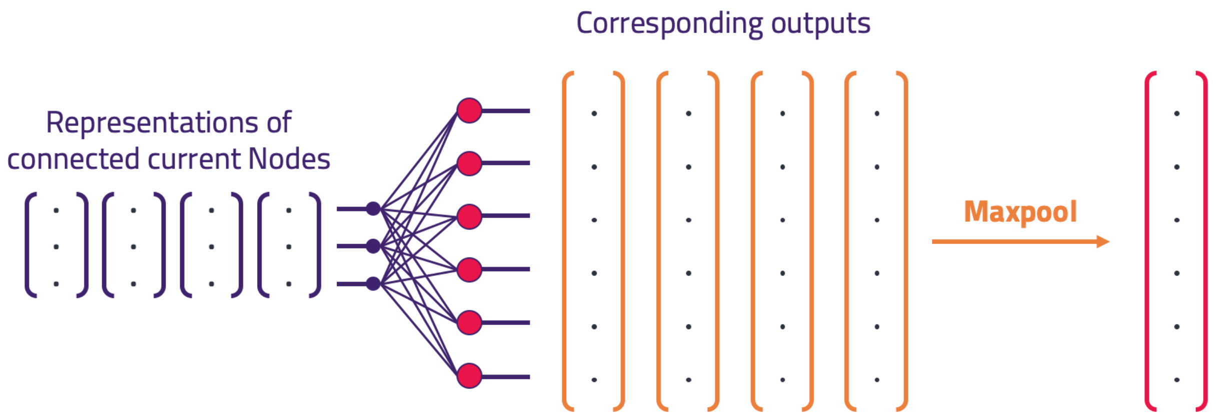


The diagram above shows how this chaining works in the case of supervised classification.

The Embedder is responsible for chaining the sub-components Aggregator and Combiner, explained below.

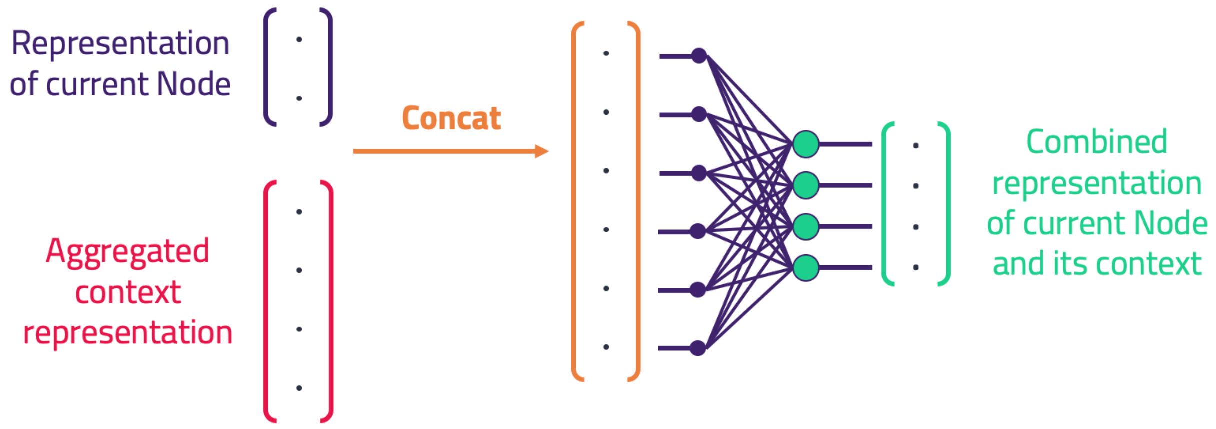
Aggregator

An *Aggregator* (pictured below) takes in a vector representation of a sub-sample of a Thing's neighbours. It produces one vector that is representative of all of those inputs. It must do this in a way that is order agnostic, since the neighbours are unordered. To achieve this we use one densely connected layer, and *maxpool* the outputs (maxpool is order-agnostic).



Combiner

Once we have Aggregated the neighbours of a Thing into a single vector representation, we need to combine this with the vector representation of that thing itself. A *Combiner* achieves this by concatenating the two vectors, and reduces the dimensionality using a single densely connected layer.


Representation of current Node
Aggregated context representation
Concat
Combined representation of current Node and its context

Supervised KGCN Classifier

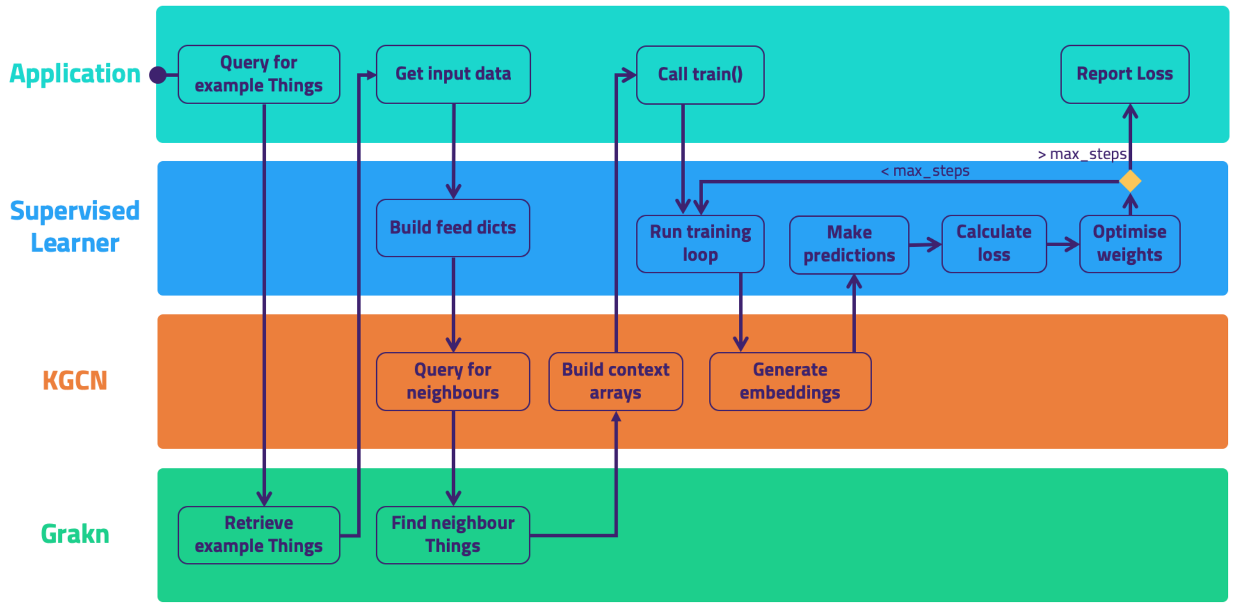
A Supervised KGCN Classifier is responsible for orchestrating the actual learning. It takes in a KGCN instance and as for any learner making use of a KGCN, it provides:

- Methods for **train/evaluation/prediction**
- **A pipeline** from embedding tensors to predictions
- **A loss function** that takes in predictions and labels

- An optimiser
- The backpropagation training loop

It must be the class that provides these behaviours, since a KGCN is not coupled to any particular learning task. This class therefore provides all of the specialisations required for a supervised learning framework.

Below is a slightly simplified UML activity diagram of the program flow.



Build with KGCNs

To start building with KGCNs, take a look at the readme's quickstart, ensure that you have all of the requirements and follow the sample usage instructions.

This will get you on your way to build a multi-class classifier for your own knowledge graph! There's also an example in the repository with real data that should fill in any gaps the template usage misses.

If you are interested in KGCNs there are several things you can do:

- Submit an issue for any problems you encounter installing or using KGCNs
- Ask questions, propose ideas or have a conversation with the Grakn team and community members on the #kglib channel on the Grakn Community slack