**Introduction**

Recommender systems are an intuitive line of defence against consumer over-choice. Given the explosive growth of information available on the web, users are often greeted with more than countless products, movies or restaurants. As such, personalization is an essential strategy for facilitating a better user experience. All in all, these systems have been playing a vital and indispensable role in various information access systems to boost business and facilitate decision-making process and are pervasive across numerous web domains such as e-commerce and/or media websites. In general, recommendation lists are generated based on user preferences, item features, user-item past interactions and some other additional information such as temporal (e.g., sequence-aware recommender) and spatial (e.g., POI recommender) data. Recommendation models are mainly categorized into collaborative filtering, content-based recommender system and hybrid recommender system based on the types of input data. Deep learning enjoys a massive hype at the moment. The past few decades have witnessed the tremendous success of the deep learning (DL) in many application domains such as computer vision and speech recognition. The academia and industry have been in a race to apply deep learning to a wider range of applications due to its capability in solving many complex tasks while providing start-of-the-art results. Recently, deep learning has been revolutionizing the recommendation architectures dramatically and brings more opportunities to improve the performance of recommender. Recent advances in deep learning-based recommender systems have gained significant attention by overcoming obstacles of conventional models and achieving high recommendation quality. Deep learning is able to effectively capture the non-linear and non-trivial user-item relationships, and enable the codification of more complex abstractions as data representations in the higher layers. Furthermore, it catches the intricate relationships within the data itself, from abundant accessible data sources such as contextual, textual and visual information.

**Pervasiveness and ubiquity of deep learning in recommender systems**. In industry, recommender systems are critical tools to enhance user experience and promote sales/services for many online websites and mobile. For example, 80 percent of movies watched on Netflix came from recommendations, 60 percent of video clicks came from home page recommendation in YouTube. Recently, many companies employ deep learning for further enhancing their recommendation quality. Covington et al. presented a deep neural network based recommendation algorithm for video recommendation on YouTube. Cheng et al. proposed an App recommender system for Google Play with a wide & deep model. Shumpei et al. presented a RNN based news recommender system for Yahoo News. All of these models have stood the online testing and shown significant improvement over traditional models. Thus, we can see that deep learning has driven a remarkable revolution in industrial recommender applications. The number of research publications on deep learning based recommendation methods has increased exponentially in these years, providing strong evidence of the inevitable pervasiveness of deep learning in recommender system research. The leading international conference on recommender system, RecSys, started to organize regular workshop on deep learning for recommender system since the year 2016. This workshop aims to promote research and encourage applications of deep learning-based recommender system. The success of deep learning for recommendation both in academia and in industry requires a comprehensive review and summary for successive researchers and practitioners to better understand the strength and weakness, and application scenarios of these models.

**Deep Learning for Recommender System**

So now understand the theory behind deep learning and neural networks,

let's finally discuss about how these techniques can be applied to Recommender systems. We can imagine that with all the buzz around artificial intelligence lately,

there's a lot of current research on applying neural networks to Recommender systems. But is it all hype? Let's get something out of the way right up front.

Just because the new technology is hot, doesn't mean that it's the right solution for every problem. We can train a neural network with user ratings or purchases and use it to make recommendations, As we've seen, deep learning can be very good at recognizing patterns in a way similar to how our brain may do it.

It's good at things like image recognition and predicting sequences of events

but is making recommendations really a pattern recognition problem?

**Deep Learning Techniques:**

Deep learning can be generally considered to be sub-€eld of machine learning. The typical defining essence of deep learning is that it learns deep representations, i.e., learning multiple levels of representations and abstractions from data. For practical reasons, we consider any neural differentiable architecture as ‘deep learning ‘as long as it optimizes a differentiable objective function using a variant of stochastic gradient descent (SGD). Neural architectures have demonstrated tremendous success in both supervised and unsupervised learning tasks. In this subsection, we clarify a diverse array of architectural paradigms that are closely related to this survey.

* **Restricted Boltzmann Machine (RBM)** is a two layer neural network consisting of a visible layer and a hidden layer. It can be easily stacked to a deep net. Restricted here means that there are no intra-layer communications in visible layer or hidden layer.
* **Autoencoder (AE)** is an unsupervised model a.empting to reconstruct its input data in the output layer. In general, the bo.leneck layer (the middle-most layer) is used as a salient feature representation of the input data. .ere are many variants of autoencoders such as denoising autoencoder, marginalized denoising autoencoder, sparse autoencoder, contractive autoencoder and variational autoencoder (VAE).
* **Multilayer Perceptron (MLP)** is a feed-forward neural network with multiple (one or more) hidden layers between the input layer and output layer. Here, the perceptron can employ arbitrary activation function and does not necessarily represent strictly binary classi€er. MLPs can be intrepreted as stacked layers of nonlinear transformations, learning hierarchical feature representations. MLPs are also known to be universal approximators.

**Why Deep Neural Networks for Recommendation?**

Before diving into the details of recent advances, it is beneficial to understand the reasons of applying deep learning techniques to recommender systems. It is evident that numerous deep recommender systems have been proposed in a short span of several years. The field is indeed bustling with innovation. At this point, it would be easy to question the need for so many different architectures and/or possibly even the utility of neural networks for the problem domain. Along the same tangent, it would be apt to provide a clear rationale of why

each proposed architecture and to which scenario it would be most beneficial for. All in all, this question is highly relevant to the issue of task, domains and recommender scenarios. One of the most attractive properties of neural architectures is that they are (1) end-to-end differentiable and (2) provide suitable inductive biases catered to the input data type. As such, if there is an inherent structure that the model can exploit, then deep neural networks

ought to be useful. For instance, CNNs and RNNs have long exploited the intrinsic structure in vision (and/or human language). Similarly, the sequential structure of session or click-logs are highly suitable for the inductive biases provided by recurrent/convolutional models.

To recapitulate, we summarize the strengths of deep learning-based recommendation models that readers might bear in mind when try to employ them for practice use.

**Nonlinear Transformation**. Contrary to linear models, deep neural networks is capable of modelling the non-linearity in data with nonlinear activations such as relu, sigmoid, tanh, etc. This property makes it possible to capture the complex and intricate user item interaction patterns.

**Representation Learning**. Deep neural networks is efficacious in learning the underlying explanatory factors and useful representations from input data. In general, a large amount of descriptive information about items and users is available in real-world applications.

Sequence Modelling. Deep neural networks have shown promising results on a number of sequential modelling tasks such as machine translation, natural language understanding, speech recognition, chatbots, and many others. RNN and CNN play critical roles in these tasks. RNN achieves this with internal memory states while CNN achieves this with filters sliding along with time.

Flexibility. Deep learning techniques possess high flexibility, especially with the advent of many popular deep learning frameworks such as TensorFlow, Keras, Caffe, MX net, DeepLearning4j, PyTorch, Theano etc.

**Potential Limitations**

Are there really any drawbacks and limitations with using deep learning for recommendation? In this section, we aim to tackle several commonly cited arguments against the usage of deep learning for recommender systems research.

• **Interpretability**. Despite its success, deep learning is well-known to behave as black boxes, and providing explainable predictions seem to be a really challenging task. A common argument against deep neural networks is that the hidden weights and activations are generally non-interpretable, limiting explain ability. However, this concern has generally been eased with the advent of neural attention models and have paved the world for deep neural models that enjoy improved interpretability. While interpreting individual neurons still pose a challenge for neural models (not only in recommender systems), present state-of-the-art models are already capable of some extent of interpretability, enabling explainable recommendation.

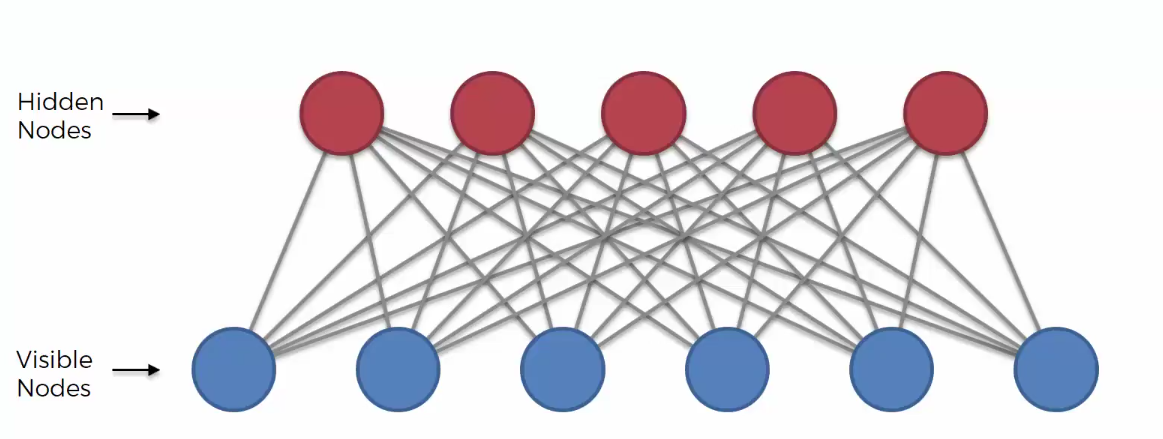
**• Data Requirement.** A second possible limitation is that deep learning is known to be data-hungry, in the sense that it requires sufficient data in order to fully support its rich parameterization. However, as compared with other domains (such as language or vision) in which labelled data is scarce, it is relatively easy to garner a significant amount of data within the context of recommender systems research. Million/billion scale datasets are commonplace not only in industry but also released as academic datasets.

**• Extensive Hyperparameter Tuning.** A third well-established argument against deep learning is the need for extensive hyperparameter tuning. However, we note that hyperparameter tuning is not an exclusive problem of deep learning but machine learning in general (e.g., regularization factors and learning rate similarly have to be tuned for traditional matrix factorization etc) Granted, deep learning may introduce additional hyperparameters in some cases. For example, a recent work, attentive extension of the traditional metric learning algorithm only introduces a single hyperparameter.

**Restricted Boltzmann Machines (RBM's):**

A **Boltzmann Machine** is a network of symmetrically connected, neuron-like units that make stochastic decisions about whether to be on or off. Boltzmann machines have a simple learning algorithm that allows them to discover interesting features in datasets composed of binary vectors. The learning algorithm is very slow in networks with many layers of feature detectors, but it can be made much faster by learning one layer of feature detectors at a time.

A **Restricted Boltzmann Machine** (RBM) is a specific type of a Boltzmann machine, which has two layers of units. As illustrated below, the first layer consists of visible units, and the second layer includes hidden units. In this restricted architecture, there are no connections between units in a layer.



The visible units in the model correspond to the observed components, and the hidden units represent the dependencies between these observed components. The goal is to model a joint probability of visible and hidden units: **p(v, h)**. Because there are no connections between hidden units, the learning is effective as all hidden units are conditionally independent, given the visible units.

Restricted Boltzmann Machine, or RBM for short. It's been in use since 2007, long before AI had its big resurgence but it's still a commonly cited paper and a technique that's still in use today. Going back to the Netflix Prize, the main things Netflix learned was that matrix factorization and RBM's had the best performance as measured by RMSE, and their scores were almost identical. Again, this shouldn't surprise us too much since we know that you can model matrix factorization as a neural network. But they found that by combining matrix factorization with RBM's, the two of them working together provided even better results, they went from an RMSE of 8.9 to 8.8. A few years ago, Netflix confirmed they were still using RBM's as part of their recommender system that's in production.

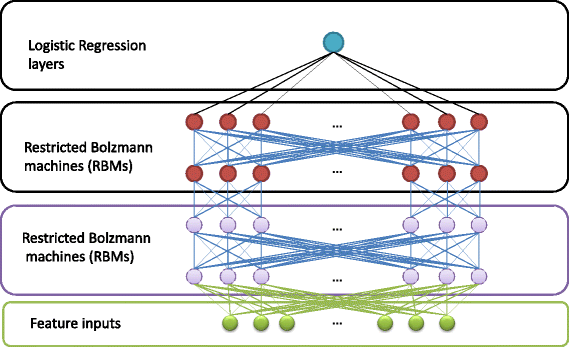
RBM's are really one of the simplest neural networks, it's just two layers, a visible layer and a hidden layer. We train it by feeding our training data into the visible layer in a forward pass, and training weights and biases between them during backpropagation. An activation function such as ReLU is used to produce

the output from each hidden neuron.

**Why are they called Restricted Boltzmann Machines?**

Well, they are restricted because neurons in the same layer can't communicate with each other directly . There are only connections between the two different layers. but that restriction didn't exist with earlier Boltzmann Machines back when AI was still kind of floundering as a field. And RBM's weren't invented by a guy named Boltzmann, the name refers to the Boltzmann distribution function they used for their sampling function.

A **Deep Belief Network** (DBN)is a multi-layer learning architecture that uses a stack of RBMs to extract a deep hierarchical representation of the training data. In such a design, the hidden layer of each sub-network serves as the visible layer for the upcoming sub-network.



When learning through a DBN, firstly, the RBM in the bottom layer is trained by inputting the original data into the visible units. Then, the parameters are fixed up, and the hidden units of the RBM are used as the input into the RBM in the second layer. The learning process continues until reaching the top of the stacked sub-networks, and finally, a suitable model is obtained to extract features from the input. Since the learning process is unsupervised, it is common to add a new network of supervised learning to the end of the DBN to use it in a supervised learning task such as classification or regression

RBM's are actually credited to Geoffrey Hinston who was a professor at Carnegie Mellon university at the time, the idea actually dates back to 1985. So, RBM's get trained by doing a forward pass, which we just described, and then a backward pass, where the inputs get reconstructed. We do this iteratively over many epochs, just like when we train a deep neural network, until it converges on a set of weights and biases that minimizes the error.

**Backward Pass RBM:**

During the backward pass, The system trying to reconstruct the original input by feeding back the output of the forward pass back through the hidden layer,

and seeing what values we end up with out of the visible layer. Since those weights are initially random, there can be a big difference between the inputs we started with and the ones we reconstruct. In the process, we end up with another set of bias terms, this time on the visible layer. So, we share weights between both the forward

and backward passes, but we have two sets of biases. The hidden bias that's used in the forward pass, and the visible bias used in this backward pass.

We can then measure the error we end up with and use that information to adjust the weights a little bit during the next iteration to try and minimize that error. Conceptually, you can see it's not too different from what we call a linear threshold unit in more modern terminology. You can also construct multi-layer RBM's that are akin to modern deep learning architectures. The main difference is that we read the output of an RBM on the lower level during a backward pass, as opposed to just taking outputs on the other side like we would with a modern neural network.

So this all works well when you have a complete set of training data, for example,

applying an RBM to the same MNIST handwriting recognition problem we looked at in our deep learning intro section is a straightforward thing to do.

When you apply RBM's, or neural networks in general, to recommender systems though, things get weird. The problem is that we now have sparse training data

very sparse, in most cases.

**How do you train a neural network** when most of your input’s nodes have no data to work with? Adapting an RBM for, say, recommending movies given five-star ratings data, requires a few twists to the generic RBM architecture we just described.

Let's take a step back and think about what we're doing here. The general idea is to use each individual user in our training data as a set of inputs into our RBM to help train it. So, we process each user as part of a batch during training, looking at their ratings for every movie they rated. So, our visible nodes represent ratings

for a given user on every movie, and we're trying to learn weights and biases

to let us reconstruct ratings for user/movie pairs we don't know yet. First of all, our visible units aren't just simple nodes taking in a single input.

Ratings are really categorical data, so we actually want to treat each individual rating

as five nodes, one for each possible rating value. So, let's say the first rating we have in our training data is a five-star rating, that would be represented as four nodes with a value of zero and one with a value of one, as represented here. Then we have a couple of ratings that are missing for user/item pairs that are unknown

and need to be predicted. Then we have a three-star rating, represented like this with a one in the third slot. When we're done training the RBM, we'll have a set of weights and biases that should allow us to reconstruct ratings for any user. So to use it to predict ratings for a new user, we just run it once again using the known ratings

of the user we're interested in. We run those through in the forward pass,

then back again in the backward pass, to end up with reconstructed rating values for that user. We can then run SoftMax on each group of five rating values to translate the output back into a five-star rating for every item.

But again, the big problem is that the data we have is sparse. If we are training an RBM on every possible combination of users and movies, most of that data will be missing, because most movies have not been rated at all by a specific user. We want to predict user ratings for every movie though, so we need to leave space for all of them. That means if we have N movies, we end up with N time five

visible nodes, and for any given user, most of them are undefined and empty. We deal with this by excluding any missing ratings from processing while we're training the RBM. This is kind of a tricky thing to do, because most frameworks built for Deep Learning such as TensorFlow assume you want to process everything in parallel, all the time. Sparse data isn't something they were really built for originally, but there are ways to trick it into doing what we want. But, notice that we've only drawn lines

between visible units that actually have known ratings data in them, and the hidden layer. So as we're training our RBM with a given user's known ratings, we only attempt to learn the weights and biases used for the movies that user actually rated. As we iterate through training on all of the other users, we fill in the other weights and biases as we go. For the sake of completeness, I should point out that TensorFlow actually does have a sparse tensor these days you can use, and there are other frameworks such as Amazon's DSSTNE system that are designed to construct more typical

deep neural networks with sparse data. RBM's will probably become a thing of the past now that we can treat sparse data in the same manner as complete data with modern neural networks, and we will examine that in more depth in an upcoming section of this course. The other twist is how to best train an RBM that contains huge amounts of sparse data. Gradient descent needs a very efficient expectation function to optimize on, and for recommender systems **this function is called contrastive divergence**. At least, that's the function the paper on the topic uses successfully. The math behind it gets a bit complicated, but the basic idea is that it samples probability distributions during training using something called a Gibbs sampler. We only train it on the ratings that actually exist, but re-use the resulting weights and biases across other users to fill in the missing ratings we want to predict.

RBMs are a variant of Boltzmann machines, with the restriction that their neurons must form a bipartite graph: a pair of nodes from each of the two groups of units (commonly referred to as the "visible" and "hidden" units respectively) may have a symmetric connection between them; and there are no connections between nodes within a group. By contrast, "unrestricted" Boltzmann machines may have connections between hidden units. This restriction allows for more efficient training algorithms than are available for the general class of Boltzmann machines.

Ruslan Salakhutdinov proposed a restricted Boltzmann machine-based recommender . To the best of our knowledge, it is the first recommendation model that built on neural networks.

The visible unit of RBM is limited to binary values, therefore, the rating score is represented in a one-hot vector to adapt to this restriction.

**For example**, [0,0,0,1,0] represents that the user gives a rating score 4 to this item. Let hj , j = 1, ..., F denote the hidden units with fixed size F . Each user has a unique RBM with shared parameters. Suppose a user rated m movies, the number of visible units is m, Let X be a K ×m matrix where x y i = 1 if user u rated movie i as y and x y i = 0 otherwise. Then:

where W y ij represents the weight on the connection between the rating y of movie i and the hidden unit j, b y i is the bias of rating y for movie i, bj is the bias of hidden unit j. RBM is not tractable, but the parameters can be learned via the Contrastive Divergence (CD) algorithm . The authors further proposed using a conditional RBM to incorporate the implicit feedback. The essence here is that users implicitly tell their preferences by giving ratings, regardless of how they rate items. The above RBM-CF is user-based where a given user’s rating is clamped on the visible layer. Similarity, we can easily design an item-based RBM-CF if we clamp a given item’s rating on the visible layer.

**Auto Encoder or AutoRec: -**

Autoencoder is a special model trained to reconstruct input vectors, meaning that input and output vectors should match, despite the information bottleneck in the hidden layer. You train autoencoders to minimize this reconstruction error on unseen data vectors. The information bottleneck is implemented for example by reducing and subsequently increasing dimensionality of input vectors, by regularizing models to enforce sparsity in the hidden layer, or by a combination of such methods. In computer vision, autoencoders are successfully used to reconstruct images, where minimizing the reconstruction error can be further extended by employing a perceptual loss.

A group from the Australian National University published a paper called AutoRec:

Autoencoders Meet Collaborative Filtering, and they used the topology you see here.

It looks a lot more familiar to the sorts of networks we covered in our introduction to deep learning. You have three layers: an input layer on the bottom that contains individual ratings, a hidden layer, and an output layer that gives us our predictions.

A matrix of weights between the layers is maintained across every instance of this network, as well as a bias node for both the hidden and output layers. In the paper, they trained the network once per item, feeding in ratings from each user for that item in the input layer. A sigmoid activation function was used on the output.

All in all, it's a pretty straightforward approach, and they reported slightly better results compared to using an RBM. But the implementation is a bit different.

RBMs just had separate bias turns for each pass, while here, we have a whole separate set of weights to work with, too. This sort of architecture also has the benefit of being a lot easier to implement in modern frameworks such as Tensorflow or keras. But there's still one wrinkle: the sparsity of the data we are working with.

In the paper, they briefly mention that "we only consider the contribution of observed ratings." So, they were careful to process each path through this neural network individually, only propagating information from ratings that actually exist in the training data and ignoring the contribution from input nodes that correspond to missing data from user item pairs that weren't rated at all. This is still a tough thing to do in tensorflow. While tensorflow does have sparse tensors, there's no simple way to restrict the chain of matrix multiplications and additions needed to implement a neural network to just the input nodes with actual data in them.

Any implementation you'll find of this using tensorflow or keras just ignores that problem and models missing ratings as zeros. You can still get decent results with enough effort, but it's a very fundamental problem to applying deep learning to recommender systems. This architecture is referred to as an autoencoder.

The act of building up the weights and biases between the input and hidden layer

is referred to as encoding the input. We are encoding the patterns in the input

as a set of weights into that hidden layer. Then, as we reconstruct the output

in the weights between the hidden and output layers, we are decoding it.

So, the first set of weights is the encoding stage, and the second set is the decoding stage. Conceptually, this isn't really any different from what we were doing with RBMs. In an RBM, we encoded on the forward pass and decoded on the backward pass.

**Autoencoder based Recommendation**

There exist two general ways of applying autoencoder to recommender system: (1) using autoencoder to learn lower-dimensional feature representations at the bottleneck layer; or (2) filling the blanks of the interaction matrix directly in the reconstruction layer. Almost all the autoencoder variants such as denoising autoencoder, variational autoencoder, contractive autoencoder and marginalized autoencoder can be applied to recommendation task.

**Autoencoder based Collaborative Filtering.** One of the successful applications is to consider the collaborative filtering from Autoencoder perspective. AutoRec takes user partial vectors r (u) or item partial vectors r (i) as input, and aims to reconstruct them in the output layer. Apparently, it has two variants: item-based AutoRec (I-AutoRec) and user-based AutoRec (U-AutoRec), corresponding to the two types of inputs. Here, we only introduce I-AutoRec, while U-AutoRec can be easily derived accordingly. Given input , the reconstruction is: h(; θ) = f (W ·g(V · +µ)+b), where f (·) and g(·) are the activation functions, parameter θ = {W ,V, µ,b}. The objective function of I-AutoRec is formulated as follows:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | N |  |  |  |  |
| argmin |  | „i” | h „i”; θ | 2 +λ |  |
| θ |  | „r | ” kO | reg |

Here k · k2 O means that it only considers observed ratings. e objective function can be optimized by resilient propagation (converges faster and produces comparable results) or L-BFGS (Limited-memory Broyden Fletcher Goldfarb Shanno algorithm). ere are four important points about AutoRec that worth noticing before deployment: (1) I-AutoRec performs better than U-AutoRec, which may be due to the higher variance of user partially observed vectors. (2) Different combination of activation functions f (·) and g(·) will influence the performance considerably. (3) Increasing the hidden unit size moderately will improve the result as expanding the hidden layer dimensionality gives AutoRec more capacity to model the characteristics of the input. (4) Adding more layers to formulate a deep network can lead to slightly improvement. CFN is an extension of AutoRec, and possess the following two advantages: (1) it deploys the denoising techniques, which makes CFN more robust; (2) it incorporates the side information such as user profiles and item descriptions to mitigate the sparsity and cold start influence. e input of CFN is also partial observed vectors, so it also has two variants: I-CFN and U-CFN, taking r (i) and r (u) as input respectively. Masking noise is imposed as a strong regularize to better deal with missing elements (their values are zero). e authors introduced three widely used corruption approaches to corrupt the input: Gaussian noise, masking noise and salt-and-pepper noise. Further extension of CFN also incorporates side information. However, instead of just integrating side information in the first layer, CFN injects side information in every layer. us, the reconstruction becomes:

h({r˜ (i) , si }) = f (W2 · {д(W1 · {r (i) , si } + µ), si } + b)

where side information, {r˜ (i) , si } indicates the concatenation of r˜ (i) and si . Incorporating side information improves the prediction accuracy, speeds up the training process and enables the model to be more robust.