1. Information Sparsity
2. Deep learning is part of machine learning. How can we **get computers to learning stuff automatically**, rather than **just us having to tell them things and coding by hand** in the kind of traditional write computer program to tell it what you want it to do. **(computers learn stuffs automatically and are not instructed and coded by human)**
3. Deep learning is also profoundly different to the vast majority of what happened in machine learning in the 80s, 90s, and 00s. And this central difference is that for most of traditional machine learning, if I call it that, so this is all of the stuff like **decision trees**, **logistic repressions**, **navie bayers**, **support vector machines** and any of those sorts of things. Essentially the way that we did things was, what we did was have a human being who looked carefully at a particular problem and worked out what was important in that problem. And then designed features that would be useful features for handling the problem that they would then encoded by hand. Normally by writing little bits of Python code or something like that to recognize some features.

**(look at problems-> do data analysis -> develop theory and work out features (what are important for the properties) > code them by hand)**

1. It turns out that the machine was learning almost nothing. **So the human being was learning a lot about the problem.** They were looking at the problem hard, doing lots of data analysis, developing theory and learning a lot about what was important for this property. What was machine doing? Was **numeric optimization**. So once you had all these signals, what you are then going to be doing was building a linear classifier. Which meant that you were putting a parameter weight in front of each feature. Machine learning’s job is to adjust those numbers as to optimize performance. Computers are very good at numerical optimization. Somethings that human being less good at that. **(Computers are doing numeric optimization) (Human manually design features, build classifiers and put parameter weights in front of each feature)**
2. We find out that in practice machine learning was sort of 90% of human beings working out how to describe data and work out important features. And only sort of 10% the computer running this learning numerical optimization algorithm.
3. So in general our manually designed features tend to be over **specified, incomplete, take a long time to design and validate, and only get you to a certain level of performance at the end of the day.** Where the learned features are easy to adapt, fast to train, and they can keep on learning so that they get to a better level of performance than we have been able to achieve previously. Deep learning framework which is just great for representing all kinds of information.

**(Learning features are easy to adapt, fast to train, keep on learning so that they get to a better level of performance)**

1. Deep learning is part of this field that is called **representation learning** and the idea of representation learning is to say, **we can just feed to our computers raw signals from world**, whether that is visual signals or language signals. **And then the computers can automatically, by itself, come up with good intermediate representations that will allow it to do tasks well**. So in some sense, it is going to be invented its own features in the same way that in the past the human being was inventing features.
2. The real meaning of the word deep learning is the argument that you could **have multiple layers of learned representation**. And that you would be able to outperform other methods of learning by having **multiple layers of learned representations**.

Project types:

1. Apply existing NN model to a new task
2. Implement a complex neural architecture
3. Come up with a new NN model
4. Theory of deepen learning, for example, optimization
5. Apply existing NN model to a new task
6. Define Task
7. Define dataset

* Search for academic datasets

They already have baselines, get rid of complexity

Define your own (harder, need more new baselines) (lot of tasks of data preprocessing)

1. Define your metric

Search online for well established metrics on this task

1. Split the dataset

Train/Dev/Test

Academic dataset often come pre-split

1. Establish a baseline

Implement the simples model (often logistic regression on unigrams and bigrams) first

Compute metrics on train AND dev

Analyze errors

If metrics are amazing and no errors: done, problem too easy, restart

1. Implement existing neural net model

Compute metric on train and dev

Analyze output and errors

Minimum bar for this class

1. Always be close to your data

Visualize the dataset

Collect summary statistics

Look at errors

Analyze how different hyperparameters affect performance

1. Try out different model variants

Soon you have more options

* Fixed window neural network
* Recurrent neural network
* Recursive neural network
* Convolutional neural network

1. a complex neural architecture

* do all other steps first
* gain intuition of why existing models are flawed
* Implement new models and iterate quickly over ideas
* Set up efficient experimental framework
* Build simpler new models first
* Data preprocessing, Greedy search, implement model, stretch goals

First Major Obstacle:

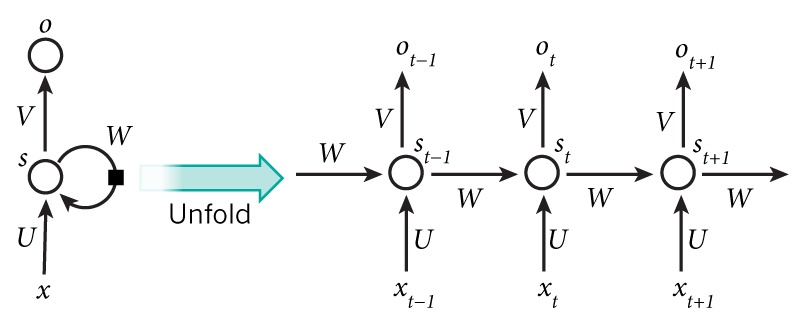
No single model architecture with consistent state of the art results across tasks

Second Major Obstacle:

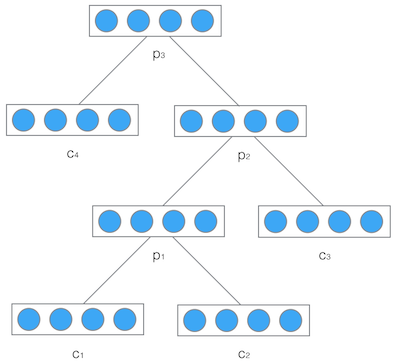
Fully joint multitask learning\* is hard

* Usually restricted to lower layers
* Usually helps only if tasks are related
* Often hurts performance if tasks are not related

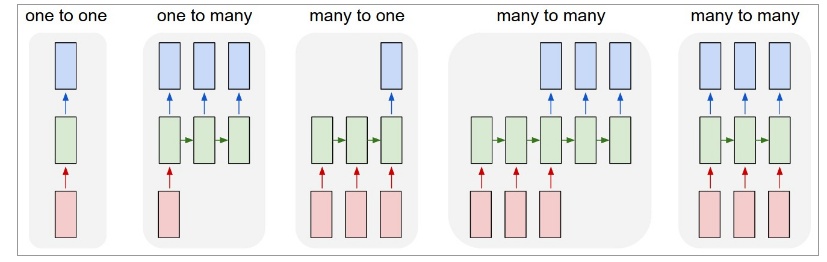
\*Meaning: share the same decoder/classifier and not only transfer learning

Recurrent neural network

Recursive neural network

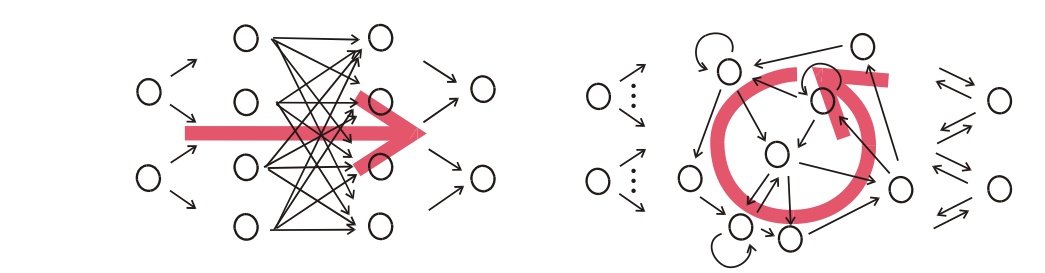


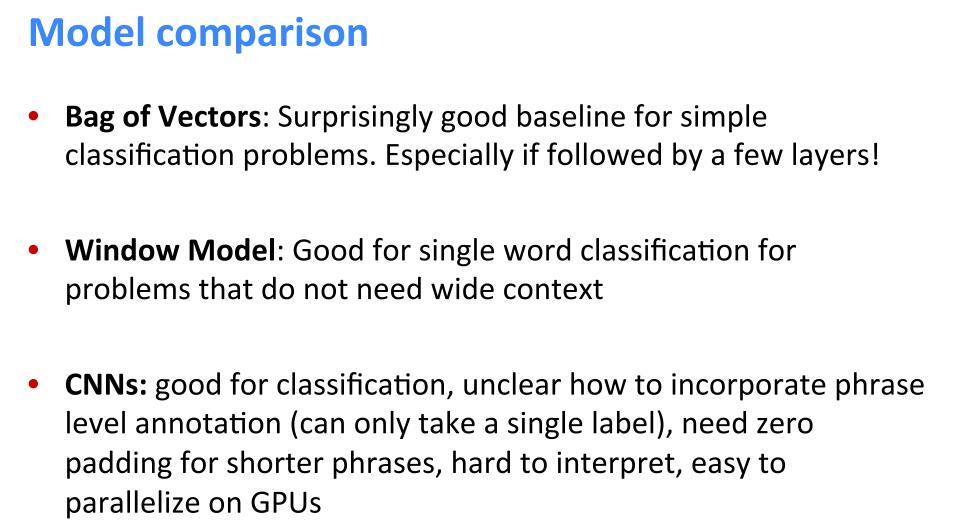
首先， 要看RNN和对于图像等静态类变量处理立下神功的卷积网络CNN的结构区别来看， “循环”两个字，已经点出了RNN的核心特征， 即系统的输出会保留在网络里， 和系统下一刻的输入一起共同决定下一刻的输出。**这就把动力学的本质体现了出来， 循环正对应动力学系统的反馈概念，可以刻画复杂的历史依赖。另一个角度看也符合著名的图灵机原理。** 即此刻的状态包含上一刻的历史，又是下一刻变化的依据。 这其实包含了可编程神经网络的核心概念，即， 当你有一个未知的过程，但你可以测量到输入和输出， 你假设当这个过程通过RNN的时候，它是可以自己学会这样的输入输出规律的， 而且因此具有预测能力。 在这点上说， RNN是图灵完备的。



图： 图1即CNN的架构， 图2到5是RNN的几种基本玩法。图2是把单一输入转化为序列输出，例如把图像转化成一行文字。 图三是把序列输入转化为单个输出， 比如情感测试，测量一段话正面或负面的情绪。 图四是把序列转化为序列， 最典型的是机器翻译， 注意输入和输出的“时差”。 图5是无时差的序列到序列转化， 比如给一个录像中的每一帧贴标签。

R**NN的重要特性是可以处理不定长的输入，得到一定的输出**。当你的输入可长可短， 比如训练翻译模型的时候， 你的句子长度都不固定，你是无法像一个训练固定像素的图像那样用CNN搞定的。而利用RNN的循环特性可以轻松搞定。





Predict unseen….

Dependence ……

Artificial intelligence – requires being able to understand bigger things (predict unseen) from knowing about smaller parts (Dependence)

Relationship between RNNS and CNN

RNN: Get compositional vectors for grammatical phrase only

CNN: Computes vectors for every possible phrase

|  |  |  |
| --- | --- | --- |
| Classifier | Features | F1 |
| SVM | POS, stemming, syntactic patterns | 60.1 |
| MaxEnt | POS. WordNet, morphological features, Noun compound system, thesauri, Google n-grams | 77.6 |
| SVM | POS, WordNet, prefixes, morphological features, dependency parse features, Levin classes, PropBank, FrameNet, NomLex-Plus, Google n-grams, paraphrases, TextRunner | 82.2 |
| RNN |  |  |
| MV-RNN |  |  |
| MV-RNN | POS, WordNet, NER | 82.4 |

How to choose an algorithm for a particular application

How to monitor and respond to feedback obtained from experiments

**Performance Metrics**

What is important is to determine which performance metric to improve ahead of time, then concentrate on improving this metric. Without clearly deﬁned goals, it can be diﬃcult to tell whether changes to a machine learning system make progress or not.

It is impossible to achieve absolute zero error. This is because your input features may not contain complete information about the output variable, or because the system might be intrinsically stochastic. You will also be limited by having a finite amount of training data.

In some cases, this is because it is diﬃcult to decide what should be measured. For example, when performing a transcription task, should we measure the accuracy of the system at transcribing entire sequences, or should we use a more ﬁne-grained performance measure that gives partial credit for getting some elements of the sequence correct? When performing a regression task, should we penalize the system more if it frequently makes medium-sized mistakes or if it rarely makes very large mistakes? These kinds of design choices depend on the application.

In

Sometimes it is much more costly to make one kind of a mistake than another. Rather than measuring the error rate of a spam classiﬁer, we may wish to measure some form of total cost, where the cost of locking legitimate messages is higher than the cost of allowing spam messages.

In many cases, we wish to summarize the performance of the classifier with a single number rather than a curve. To do so, we can convert precision p and recall r into an **F-score**

* In some applications, it is possible for the machine learning system to refuse to make a decision. This is useful when the machine learning algorithm can estimate how confident it should be about a decision, especially if a wrong decision can be harmful and if a human operator is able to occasionally take over.
* A natural performance metric to use in this situation is coverage. Coverage is the fraction of examples for which the machine learning system is able to produce a response. It is possible to trade coverage for accuracy. One can always obtain 100 percent accuracy by refusing to process any example, but this reduces the coverage to 0 percent.

**Selecting Models**

you may want to begin with a simple statistical model like logistic regression

If your problem has a chance of being solved by just choosing a few linear weights correctly

If you know that your problem falls into an “AI-complete” category like object recognition, speech recognition, machine translation, and so on.

you are likely to do well by beginning with an appropriate deep learning model.

* fixed-size vectors as input-> feedforward network with fully connected layers
* the input has known as topological structure (for example, if the input is an image) -> use a convolutional network
* the input or output is a sequence-> use a gated recurrent net

First, choose the general category of model based on the structure of your

data.

If your task is similar to another task that has been studied extensively, you will probably do well by ﬁrst copying the model and algorithm that is already known to perform best on the previously studied task. You may even want to copy a trained model from that task.

Some domains, such as natural language processing, are known to benefit tremendously from unsupervised learning techniques, such as learning unsupervised word embeddings. In other domains, such as computer vision, current unsupervised learning techniques do not bring a benefit, except in the semi-supervised setting, when the number of labeled examples is very small.

only use unsupervised learning in your first attempt if the task you want to solve is unsupervised. You can always try adding unsupervised learning later if you observe that your initial

baseline overfits.

**Whether to Gather More Data**

First, determine whether the performance on the training set is acceptable. If performance on the training set is poor, the learning algorithm is not using the training data that is already available, so there is no reason to gather more data.

1. Instead, try increasing the size of the model by adding more layers or adding more hidden units to each layer. Also, try improving the learning algorithm, for example by tuning the learning rate hyperparameter. -> make training set work rather than gather more data
2. If large models and carefully tuned optimization algorithms do not work well, then the problem might be the quality of the training data. The data may be too noisy or may not include the right inputs needed to predict the desired outputs. This suggests starting over, collecting cleaner data, or collecting a richer set of features.

If the performance on the training set is acceptable, then measure the performance on a test set. If the performance on the test set is also acceptable, then there is nothing left to be done.

If test set performance is much worse than training set performance, then gathering more data is one of the most effective solutions. The key considerations are the cost and feasibility of gathering more data, the cost and feasibility of reducing the test error by other means, and the amount of data that is expected to be necessary to improve test set performance significantly.

A simple alternative to gathering more data is to reduce the size of the model or improve regularization, by adjusting hyperparameters such as weight decay coefficients, or by adding regularization strategies such as dropout. If you find that the gap between train and test performance is still unacceptable even after tuning the regularization hyperparameters, then gathering more data is advisable.

When deciding whether to gather more data, it is also necessary to decide how much to gather. plot curves showing the relationship between training set size and generalization error. Usually, adding a small fraction of the total number of examples will not have a noticeable effect on generalization error. It is therefore recommended to experiment with training set sizes on a logarithmic scale, for example, doubling the number of examples between consecutive experiments.

If gathering much more data is not feasible, the only other way to improve generalization error is to improve the learning algorithm itself.

Selecting Hyperparameters: relationship between hyperparameters, training error, generalization error and computational resources (memory and runtime)

Manual Hyperparameter Tuning

Automatic Hyperparameter Optimization Algorithms

Increasing the number of

hidden units increases the

representational capacity

of the model.

Number of hidden

units

A wider kernel results in a narrower output dimension, reducing model capacity unless you use implicit zero padding to

reduce this effect. Wider kernels require more memory for parameter storage

and increase runtime, but a narrower output reduces memory cost.

Convolution kernel width

Increasing the kernel width increases the number of parameters in the model.

Implicit zero

padding

Adding implicit zeros be- fore convolution keeps the representation size large.

An improper learning rate,

whether too high or too

low, results in a model

with low effective capacity

due to optimization failure.

Learning rate

Decreasing the weight decay

coefficient frees the

model parameters to become

larger.

Weight decay coefficient

Dropout rate

Dropping units less often

gives the units more opportunities to “conspire” with each other to fit the training set.

to develop algorithms that wrap a learning algorithm and choose its hyperparameters, thus hiding the hyperparameters of the learning algorithm from the user. Unfortunately, hyperparameter optimization algorithms often have their own hyperparameters, such as the range of values that should be explored for each of the learning algorithm’s hyperparameters. These secondary hyperparameters are usually easier to choose, however, in the sense that acceptable performance may be achieved on a wide range of tasks using the same secondary hyperparameters for all tasks.

Grid search: To perform grid search, we provide a set of values for each hyperparameter. The search algorithm runs training for every joint hyperparameter setting in the cross product of these sets.

In the case of numerical (ordered) hyperparameters, the smallest and largest element of each list is chosen conservatively, based on prior experience with similar experiments, to make sure that the optimal value is likely to be in the selected range. Typically, a grid search involves picking values approximately on a logarithmic scale, e.g., a learning rate taken within the set

The obvious problem with grid search is that its computational cost grows **exponentially** with the number of hyperparameters. If there are m hyperparameters, each taking at most n values, then the number of training and evaluation trials required grows as O(nm ). The trials may be run in parallel and exploit loose parallelism (with almost no need for communication between diﬀerent machines carrying out the search).

Random search: A random search proceeds as follows. First we deﬁne a marginal distribution for each hyperparameter, for example, a Bernoulli or multinoulli for binary or discrete hyperparameters, or a uniform distribution on a log-scale for positive real-valued hyperparameters. Usually most of these hyperparameters are independent from each other. Common choices for the distribution over a single hyperparameter include uniform and log-uniform (to sample from a log-uniform distribution, take the exp of a sample from a uniform distribution). The search algorithm then randomly samples joint hyperparameter conﬁgurations and runs training with each of them.

Model-Based Hyperparameter Optimization

The search for good hyperparameters can be cast as an optimization problem. The decision variables are the hyperparameters. The cost to be optimized is the validation set error that results from training using these hyperparameters.

Most model-based algorithms for hyperparameter search use a Bayesian regression model to estimate both the **expected value** of the **validation set error** for each hyperparameter and the **uncertainty** around this expectation. Optimization thus involves a trade-oﬀ between **exploration** (proposing hyperparameters for that there is high uncertainty, which may lead to a large improvement but may also perform poorly) and **exploitation** (proposing hyperparameters that the model is conﬁdent will perform as well as any hyperparameters it has seen so far—usually hyperparameters that are very similar to ones it has seen before). Contemporary approaches to hyperparameter optimization include Spearmint (Snoek et al., 2012), TPE, MAC.

One drawback common to most hyperparameter optimization algorithms with more sophistication than random search is that they require for a training experiment to run to completion before they are able to extract any information from the experiment. This is much less eﬃcient, in the sense of how much information can be gleaned early in an experiment, than manual search by a human practitioner, **since one can usually tell early on if some set of hyperparameters is completely pathological**.

**Debugging Strategies**

*Visualize the model in action*

*Visualize the worst mistakes*

*Reason about software using training and test errors*

*Fit a tiny dataset*

*Compare back-propagated derivatives to numerical derivatives*

*Monitor histograms of activations and gradient*