Random Ponderings

Tuesday, January 13, 2015

A Brief Overview of Deep Learning

(This is a guest post by Ilya Sutskever on the intuition behind deep learning as well as some very useful practical advice. Many thanks to Ilya for such a heroic effort!)

Deep Learning is really popular these days. Big and small companies are getting into it and making money off it. It's hot. There is some substance to the hype, too: large deep neural networks achieve the best results on speech recognition, visual object recognition, and several language related tasks, such as machine translation and language modeling.

But why? What's so special about deep learning? (from now on, we shall use the term **Large Deep Neural Networks --- LDNN ---** which is what the vaguer term "Deep Learning" mostly refers to). Why does it work now, and how does it differ from neural networks of old? Finally, suppose you want to train an LDNN. Rumor has it that it's very difficult to do so, that it is "black magic" that requires years of experience. And while it is true that experience helps quite a bit, the amount of "trickery" is surprisingly limited ---- one needs be on the lookout for only a small number well-known pitfalls. Also, there are many open-source implementations of various state-of-the-art neural networks (c.f. Caffe, cuda-covnet, Torch, Theano), which makes it much easier to learn all the details needed to make it work.

Why Does Deep Learning Work?

It is clear that, to solve hard problems, we must use powerful models. This statement is obvious. Indeed, if a model is not powerful, then there is absolutely no chance that it can succeed in solving a hard problem, no matter how good the learning algorithm is.

The other necessary condition for success is that our model is trainable. That too is obvious, for if we cannot train our model, then its power is useless --- it will never amount to anything, and great results will not be achieved. The model will forever remain in a state of unrealized potential.

Fortunately, LDNNs are both trainable and powerful.

Why Are LDNNs Powerful?

When I talk about LDNNs, I'm talking about 10-20 layer neural networks (because this is what can be trained with today's algorithms). I can provide a few ways of looking at LDNNs that will illuminate the reason they can do as well as they do.

> Conventional statistical models learn simple patterns or clusters. In contrast, LDNNs learn computation, albeit a massively parallel computation with a modest number of steps. Indeed, this is the key difference between LDNNs and other statistical models.
>
> To elaborate further: it is well known that any algorithm can be implemented by an appropriate very deep circuit (with a layer for each timestep of the algorithm's execution -- one example). What's more, the deeper the circuit, the more expensive are the algorithms that can be implemented by the circuit (in terms of runtime). And given that neural networks are circuits as well, deeper neural networks can implement algorithms with more steps ---- which is why depth = more power.
>
> > N.B.: It is easy to see that a single neuron of a neural network can compute the conjunction of its inputs, or the disjunction of its inputs, by simply setting their connections to appropriate values.
>
> Surprisingly, neural networks are actually more efficient than boolean circuits. By

About Me

Yisong Yue

G+ Follow

I am a machine learning researcher & assistant professor at Caltech. My research interests lie primarily in the theory and practice of statistical machine learning. More broadly, I am interested in working towards general artificial intelligence while developing useful intermediate tools along the way. More information is available here.

View my complete profile

my website
my photostream
my Quora answers

Labels

- adventures
- announcements
- computer science
- education
- humor
- information retrieval
- internet / networks
- life extension
- machine learning
- math
- random ponderings
- rationality / intelligence
- science / technology

more efficient, I mean that a fairly shallow DNN can solve problems that require many more layers of boolean circuits. For a specific example, consider the highly surprising fact that a DNN with 2 hidden layer and a modest number of units can sort N N-bit numbers! I found the result shocking when I heard about it, so I implemented a small neural network and trained it to sort 10 6-bit numbers, which was easy to do to my surprise. It is impossible to sort N N-bit numbers with a boolean circuit that has two hidden layers and that are not gigantic.

> The reason DNNs are more efficient than boolean circuits is because neurons perform a threshold operation, which cannot be done with a tiny boolean circuit.

Finally, human neurons are slow yet humans can perform lots of complicated tasks in a fraction of a second. More specifically, it is well-known that a human neuron fires no more than 100 times per second. This means that, if a human can solve a problem in 0.1 seconds, then our neurons have enough time to fire only 10 times --- definitely not much more than that. It therefore follows that a large neural network with 10 layers can do anything a human can in 0.1 seconds.

This is not scientific fact since it is conceivable that real neurons are much more powerful than artificial neurons, but real neurons may also turn out to be much less powerful than artificial neurons. In any event, the above is certainly a plausible hypothesis.

This is interesting because humans can solve many complicated perception problems in 0.1 seconds --- for example, humans can recognize the identity of an object that's in front of them, recognize a face, recognize an emotion, and understand speech in a fraction of a second. In fact, if there exists even just one person in the entire world who has achieved an uncanny expertise in performing a highly complex task of some sort in a fraction of a second, then this is highly convincing evidence that a large DNN could solve the same task --- if only its connections are set to the appropriate values.

But won't the neural network need to be huge? Maybe. But we definitely know that it won't have to be exponentially large ---- simply because the brain isn't exponentially large! And if human neurons turn out to be noisy (for example), which means that many human neurons are required to implement a single real-valued operation that can be done using just one artificial neuron, then the number of neurons required by our DNNs to match a human after 0.1 seconds is greatly diminished.

These four arguments suggest (strongly, in my opinion), that for a very wide variety of problems, there exists a setting of the connections of a LDNN that basically solves the problem. Crucially, the number of units required to solve these problems is far from exponential --- on the contrary, the number of units required is often so "small" that it is even possible, using current hardware, to train a network that achieves super-high performance on the task of interest. It is this last point which is so important, and requires additional elaboration:

We know that most machine learning algorithms are consistent: that is, they will solve the problem given enough data. But consistency generally requires an exponentially large amount of data. For example, the nearest neighbor algorithm can definitely solve any problem by memorizing the correct answer to every conceivable input. The same is true for a support vector machine --- we'd have a support vector for almost every possible training case for very hard problems. The same is also true for a neural network with a single hidden layer: if we have a neuron for every conceivable training case, so that neuron fires for that training case and but not for any other, then we could also learn and represent every conceivable function from inputs to outputs. Everything can be done given exponential resources, but it is never ever going to be relevant in our limited physical universe.

And it is in this point that LDNNs differ from previous methods: we can be reasonably certain that a large but not huge LDNN will achieve good results on a surprising variety of problems that we may want to solve. If a problem can be solved by a human in a fraction of a second, then we have a very non-exponential super-pessimistic upper bound on the size of the smallest neural network that can achieve very good performance.

But I must admit that it is impossible to predict whether a given problem will be

Subscribe To

Posts

Comments

My Blog List

Turing Machine
overview/ big trends in cyber collaboration & social networking
9 hours ago

Computational Complexity
Hillary Putnam passed away on March 13
23 hours ago

Shtetl-Optimized
A postdoc post
2 days ago

Bert Huang's Blog
Scientific Hypothesis: We are the Best
2 days ago

Slate Star Codex
Book Review: The Art Of The Deal
2 days ago

Freakonomics
"I Don't Know What You've Done With My Husband But He's A Changed Man"
3 days ago

Gödel's Lost Letter and P=NP
Happy St. Patrick's Day
4 days ago

Overcoming Bias
Me in London, Oxford, Istanbul
5 days ago

The Gates Notes
My Visit to Metropolis
6 days ago

The Wronger Box
Super Hard Quiz, Vol. 9
6 days ago

Show All

Science News

NYT > Technology
Andrew S. Grove, Longtime Chief of Intel, Dies at 79
22 minutes ago

NYT > Science
Randall Munroe, XKCD Creator, Goes Back to High School
2 hours ago

ScienceDaily: Latest Science News
Many species now going extinct may vanish without a fossil trace
3 hours ago

PhysOrg.com - latest science and technology news
Archaeologists create 3-D interactive digital reconstruction of King Richard III
3 hours ago

solvable by a deep neural network ahead of time, although it is often possible to tell whenever we know that a similar problem can be solved by an LDNN of a manageable size.

So that's it, then. Given a problem, such as visual object recognition, all we need is to train a giant convolutional neural network with 50 layers. Clearly a giant convnet with 50 layers can be configured to achieve human-level performance on object recognition --- right? So we simply need to find these weights. Once once we do, the problem is solved.

Learning.

What is learning? Learning is the problem of finding a setting of the neural network's weights that achieves the best possible results on our training data. In other words, we want to "push" the information from the labelled data into the parameters so that the resulting neural network will solve our problem.

The success of Deep Learning hinges on a very fortunate fact: that well-tuned and carefully-initialized stochastic gradient descent (SGD) can train LDNNs on problems that occur in practice. It is not a trivial fact since the training error of a neural network as a function of its weights is highly non-convex. And when it comes to non-convex optimization, we were taught that all bets are off. Only convex is good, and non-convex is bad. And yet, somehow, SGD seems to be very good at training those large deep neural networks on the tasks that we care about. The problem of training neural networks is NP-hard, and in fact there exists a family of datasets such that the problem of finding the best neural network with three hidden units is NP-hard. And yet, SGD just solves it in practice. This is the main pillar of deep learning.

We can say fairly confidently that successful LDNN training relies on the "easy" correlation in the data, which allows learning to bootstrap itself towards the more "complicated" correlations in the data. I have done an experiment that seems to support this claim: I found that training a neural network to solve the parity problem is hard. I was able to train the network to solve parity for 25 bits, 29 bits, but never for 31 bits (by the way, I am not claiming that learning parity is impossible for over 30 bits --- only that I didn't succeed in doing so). Now, we know that parity is a highly unstable problem that doesn't have any linear correlations: every linear function of the inputs is completely uncorrelated with the output, which is a problem for neural networks since they are mostly linear at initialization time (so perhaps I should've used larger initial weights? I will discuss the topic of weight initialization later in the text). So my hypothesis (which is shared by many other scientists) is that neural networks start their learning process by noticing the most "blatant" correlations between the input and the output, and once they notice them they introduce several hidden units to detect them, which enables the neural network to see more complicated correlations. Etc. The process goes on. I imagine some sort of a "spectrum" of correlations --- both easy and hard, and the network jumps from a correlation to a more complicated correlation, much like an opportunistic mountain climber.

Generalization.

While it is very difficult to say anything specific about the precise nature of the optimization of neural networks (except near a local minimum where everything becomes convex and uninteresting), we can say something nontrivial and specific about generalization.

And the thing we can say is the following: in his famous 1984 paper called "A Theory of the Learnable", Valiant proved, roughly speaking, that if you have a finite number of functions, say N, then every training error will be close to every test error once you have more than log N training cases by a small constant factor. Clearly, if every training error is close to its test error, then overfitting is basically impossible (overfitting occurs when the gap between the training and the test error is large). (I am also told that this result was given in Vapnik's book as small exercise). This theorem is easy to prove but I won't do it here.

But this very simple result has a genuine implication to any implementation of neural networks. Suppose I have a neural network with N parameters. Each parameter will be a float32. So a neural network is specified with 32N bits, which means that we have no more than 2^{32N} distinct neural networks, and probably much less. This means that we won't overfit much once we have more than 32N training cases. Which is nice. It means that it's theoretically OK to count parameters. What's more, if we are quite confident that each weight only requires 4 bits (say), and that everything else is just noise, then we can be fairly confident that the number of training cases will be a small constant factor of 4N rather than 32N.

SA **Scientific American**
Cuba-U.S. Thaw Should Ease Scientific Collaborations
6 hours ago

Show All

The Conclusion:

If we want to solve a hard problem we probably need a LDNN, which has many parameters. So we need a large high-quality labelled training set to make sure that it has enough information to specify all the network's connections. And once we get that training set, we should run SGD on it until the network solves the problem. And it probably will, if our neural network is large and deep.

What Changed Since the 80s?

In the old days, people believed that neural networks could "solve everything". Why couldn't they do it in the past? There are several reasons.

> **Computers were slow.** So the neural networks of past were tiny. And tiny neural networks cannot achieve very high performance on anything. In other words, small neural networks are not powerful.
>
> **Datasets were small.** So even if it was somehow magically possible to train LDNNs, there were no large datasets that had enough information to constrain their numerous parameters. So failure was inevitable.
>
> **Nobody knew how to train deep nets.** Deep networks are important. The current best object recognition networks have between 20 and 25 successive layers of convolutions. A 2 layer neural network cannot do anything good on object recognition. Yet back in the day everyone was very sure that deep nets cannot be trained with SGD, since that would've been too good to be true!

It's funny how science progresses, and how easy it is to train deep neural networks, especially in retrospect.

Practical Advice.

Ok. So you're sold. You're convinced that LDNNs are the present and the future and you want to train it. But rumor has it that it's so hard, so difficult… or is it? The reality is that it used to be hard, but now the community has consolidated its knowledge and realized that training neural networks is easy as long as you keep the following in mind.

Here is a summary of the community's knowledge of what's important and what to look after:

> **Get the data:** Make sure that you have a high-quality dataset of input-output examples that is large, representative, and has relatively clean labels. Learning is completely impossible without such a dataset.
>
> **Preprocessing:** it is essential to center the data so that its mean is zero and so that the variance of each of its dimensions is one. Sometimes, when the input dimension varies by orders of magnitude, it is better to take the log(1 + x) of that dimension. Basically, it's important to find a faithful encoding of the input with zero mean and sensibly bounded dimensions. Doing so makes learning work much better. This is the case because the weights are updated by the formula: change in w_{ij} \propto x_idL/dy_j (w denotes the weights from layer x to layer y, and L is the loss function). If the average value of the x's is large (say, 100), then the weight updates will be very large and correlated, which makes learning bad and slow. Keeping things zero-mean and with small variance simply makes everything work much better.
>
> **Minibatches:** Use minibatches. Modern computers cannot be efficient if you process one training case at a time. It is vastly more efficient to train the network on minibatches of 128 examples, because doing so will result in massively greater throughput. It would actually be nice to use minibatches of size 1, and they would probably result in improved performance and lower overfitting; but the benefit of doing so is outweighed the massive computational gains provided by minibatches. But don't use very large minibatches because they tend to work less well and overfit more. So the practical recommendation is: use the smaller minibatch that runs efficiently on your machine.
>
> **Gradient normalization:** Divide the gradient by minibatch size. This is a good idea because of the following pleasant property: you won't need to change the learning rate (not too much, anyway), if you double the minibatch size (or halve

it).

Learning rate schedule: Start with a normal-sized learning rate (LR) and reduce it towards the end.

- A typical value of the LR is **0.1**. Amazingly, 0.1 is a good value of the learning rate for a large number of neural networks problems. Learning rates frequently tend to be smaller but rarely much larger.
- Use a **validation set** ---- a subset of the training set on which we don't train --- to decide when to lower the learning rate and when to stop training (e.g., when error on the validation set starts to increase).
- A practical suggestion for a learning rate schedule: if you see that you stopped making progress on the validation set, divide the LR by 2 (or by 5), and keep going. Eventually, the LR will become very small, at which point you will stop your training. Doing so helps ensure that you won't be (over-)fitting the training data at the detriment of validation performance, which happens easily and often. Also, lowering the LR is important, and the above recipe provides a useful approach to controlling via the validation set.

But most importantly, worry about the **Learning Rate**. One useful idea used by some researchers (e.g., Alex Krizhevsky) is to monitor the ratio between the update norm and the weight norm. This ratio should be at around 10^{-3}. If it is much smaller then learning will probably be too slow, and if it is much larger then learning will be unstable and will probably fail.

Weight initialization. Worry about the random initialization of the weights at the start of learning.

- If you are lazy, it is usually enough to do something like 0.02 * randn(num_params). A value at this scale tends to work surprisingly well over many different problems. Of course, smaller (or larger) values are also worth trying.
- If it doesn't work well (say your neural network architecture is unusual and/or very deep), then you should initialize each weight matrix with the init_scale / sqrt(layer_width) * randn. In this case init_scale should be set to 0.1 or 1, or something like that.
- Random initialization is super important for deep and recurrent nets. If you don't get it right, then it'll look like the network doesn't learn anything at all. But we know that neural networks learn once the conditions are set.
- Fun story: researchers believed, for many years, that SGD cannot train deep neural networks from random initializations. Every time they would try it, it wouldn't work. Embarrassingly, they did not succeed because they used the "small random weights" for the initialization, which works great for shallow nets but simply doesn't work for deep nets at all. When the nets are deep, the many weight matrices all multiply each other, so the effect of a suboptimal scale is amplified.
- But if your net is shallow, you can afford to be less careful with the random initialization, since SGD will just find a way to fix it.

You're now informed. Worry and care about your initialization. Try many different kinds of initialization. This effort will pay off. If the net doesn't work at all (i.e., never "gets off the ground"), keep applying pressure to the random initialization. It's the right thing to do.

If you are training RNNs or LSTMs, **use a hard constraint** over the norm of the gradient (remember that the gradient has been divided by batch size). Something like 15 or 5 works well in practice in my own experiments. Take your gradient, divide it by the size of the minibatch, and check if its norm exceeds 15 (or 5). If it does, then shrink it until it is 15 (or 5). This one little trick plays a huge difference in the training of RNNs and LSTMs, where otherwise the exploding gradient can cause learning to fail and force you to use a puny learning rate like 1e-6 which is too small to be useful.

Numerical gradient checking: If you are not using Theano or Torch, you'll be probably implementing your own gradients. It is easy to make a mistake when we implement a gradient, so it is absolutely critical to use numerical gradient checking. Doing so will give you a complete peace of mind and confidence in your code. You will know that you can invest effort in tuning the hyperparameters (such as the learning rate and the initialization) and be sure that your efforts are channeled in the right direction.

If you are using LSTMs and you want to train them on problems with very long range dependencies, you should initialize the biases of the forget gates of the LSTMs to large values. By default, the forget gates are the sigmoids of their total input, and when the weights are small, the forget gate is set to 0.5, which is adequate for some but not all problems. This is the one non-obvious caveat about the initialization of the LSTM.

Data augmentation: be creative, and find ways to algorithmically increase the number of training cases that are in your disposal. If you have images, then you should translate and rotate them; if you have speech, you should combine clean speech with all types of random noise; etc. Data augmentation is an art (unless you're dealing with images). Use common sense.

Dropout. Dropout provides an easy way to improve performance. It's trivial to implement and there's little reason to not do it. Remember to tune the dropout probability, **and to not forget to turn off Dropout and to multiply the weights by** (namely by 1-dropout probability) **at test time**. Also, be sure to train the network for longer. Unlike normal training, where the validation error often starts increasing after prolonged training, dropout nets keep getting better and better the longer you train them. So be patient.

Ensembling. Train 10 neural networks and average their predictions. It's a fairly trivial technique that results in easy, sizeable performance improvements. One may be mystified as to why averaging helps so much, but there is a simple reason for the effectiveness of averaging. Suppose that two classifiers have an error rate of 70%. Then, when they agree they are right. But when they disagree, one of them is often right, so now the average prediction will place much more weight on the correct answer. The effect will be especially strong whenever the network is confident when it's right and unconfident when it's wrong.

I am pretty sure that I haven't forgotten anything. The above 13 points cover literally everything that's needed in order to train LDNNs successfully.

So, to Summarize:

LDNNs are powerful.
LDNNs are trainable if we have a very fast computer.
So if we have a very large high-quality dataset, we can find the best LDNN for the task.
Which will solve the problem, or at least come close to solving it.

The End.

But what does the future hold? Predicting the future is obviously hard, but in general, models that do even more computation will probably be very good. The Neural Turing Machine is a very important step in this direction. Other problems include unsupervised learning, which is completely mysterious and incomprehensible in my opinion as of 8 Jan 2015. Learning very complicated "things" from data without supervision would be nice. All these problems require extensive research.

Posted by Yisong Yue at 10:22 PM
Labels: computer science, machine learning, science / technology

27 comments:

Unknown said...

Very nice and useful summary. However, I would like to argue that the reasons stated for the success of deep neural nets would apparently work as well for kernel machines (if you were to train them using an efficient algorithm that is not quadratic in the number of examples, and it is not hopeless to achieve that). In particular, I believe that deep learning offers something specific in terms of generalization, something that a general-purpose kernel such as the Gaussian kernel does not offer. I have written about the expressive power of deep nets (see recent Montufar et al NIPS paper, for example) and about their ability to generalize far from the training examples (NIPS'2005 and my 2009 book and 2013 PAMI review), which a Gaussian kernel machine or a decision tree could not do. The latter may need an exponentially large number of training examples to get the same generalization error as some deep net (exponential in depth). Now this is associated with priors that are coming with the deep net (there is no free lunch and it will not work for *any* function), such as the priors associated with distributed representations (the existence of underlying factors that generate the data, so that one can learn about each factor without having to observe all the configurations of values of all the others) and with depth (the assumption of a hierarchy of factors). To summarize, I believe that the reasons for deep nets to work go beyond what was stated in this nice post.

1/14/2015 7:32 PM

Yoshua Bengio said...

I forgot to sign the above comment: Yoshua Bengio

1/14/2015 7:37 PM

Ilya Sutskever said...

Thanks!

I think that we are expressing similar concepts in different words, and I suspect that we use the term generalization in slightly different ways.

I don't see a particular difference between a shallow net with a reasonable number of neurons and a kernel machine with a reasonable number of support vectors (its not useful to consider Kernel machines with exponentially many support vectors just like there isn't a point in considering the universal approximation theorem as both require exponential resources) --- both of these models are nearly identical, and thus equally unpowerful. Both of these models will be inferior to an LDNN with a comparable number of parameters precisely because the LDNN can do computation and the shallow models cannot. The LDNN can sort, do integer-multiplication, compute analytic functions, decompose an input into small pieces and recombine it later in a higher level representation, partition the input space into an exponential number of non-arbitrary tiny regions, etc. Ultimately, if the LDNN has 10,000 layers, then it can, in principle, execute any parallel algorithm that runs in fewer than 10,000 steps, giving this LDNN an incredible expressive power. Thus, I don't think that the argument in the article suggests that huge kernel machines should be able to solve these hard problems --- they would need to have exponentially many support vectors.

Although I didn't define it in the article, generalization (to me) means that the gap between the training and the test error is small. So for example, a very bad model that has similar training and test errors does not overfit, and hence generalizes, according to the way I use these concepts. It follows that generalization is easy to achieve whenever the capacity of the model (as measured by the number of parameters or its VC-dimension) is limited --- we merely need to use more training cases than the model has parameters / VC dimension. Thus, the difficult part is to get a low training error.

Now why is it that models that can do computation are in some sense "right" compared to models that cannot? Why is the inductive bias captured by an LDNN "good", or even "correct"? Why do LDNNs succeed on the natural problems that we often want to solve in practice? I think that it is a very nontrivial fact about the universe, and is a bit like asking "why are typical recognition problems solvable by an efficient computer program". I don't know the answer but I have two theories: 1) if they weren't solvable by an efficient computer program, then humans and animals wouldn't be solving them in the first place; and 2) there is something about the nature of physics and possibly even evolution that gives raise to problems that can usually be solvable by efficient algorithms. But that is idle speculation on my part.

1/14/2015 8:19 PM

Yoshua Bengio said...

Ok.

It is what you are calling 'powerful' which encompasses my notion of 'better prior which gives rise to better generalization' (in addition to having enough capacity). I am not 100% sure why this prior works so well, but I have a theory, which I have expressed in various papers. The idea is that the data was generated by different factors, almost independently. This is what allows you to learn about the effect of one of the factors, without having to know everything about all the other factors and their exponentially large number of interactions. For example, I see images of people, and one factor is gender, another is the hair color, and other is age, and another is wearing glasses. You see that you can build a detector for each of these factors without really needing to see all the configurations of all the other factors in the data. This assumption is really equivalent to saying that a good representation is a distributed one, in terms of having a good generalization from few examples. But to have to right representation, you need enough depth (think about the depth needed, in my example with images of persons), to extract these almost independent factors or features. So you need deep distributed representations. This assumption also arises naturally if you first assume something that appears very straightforward: the input data we observe are the effects of some underlying causes, and these causes are marginally related to each other in simple ways (e.g. independent causes being the extreme case), while the things to predict are more directly connected to causes (whereas the inputs are effects). This assumption also suggests that unsupervised pre-training and semi-supervised learning of representations will work well, and they do (when there is not enough labeled data for supervised learning to do it all). It would make sense that brains have evolved to find the causes of what we observe, by the way...

Another note: in principle (and maybe not in practice), I argue that even a shallow neural net has a potentially very serious advantage over a Gaussian kernel SVM. Some functions representable very efficiently by the neural net can require an exponential number of support vectors for the kernel SVM (this idea is found in several papers, including in the most crisp way in our last NIPS paper on the number of regions associated with shallow and deep rectifier nets).

-- Yoshua Bengio

1/14/2015 8:37 PM

Ilya Sutskever said...

I basically agree with everything. Abstract independent hidden factors of real data is unquestionably part of the explanation as to why regular deep nets succeed as well as they do in practice. At the same time, I think that independent hidden factors are not the whole story, and that there may be models whose representations will so different from the ones we are dealing with now that we may not think of them as of conventional distributed representations at all (although they will necessarily be distributed, strictly speaking).

1/14/2015 8:59 PM

Olivier Grisel said...

Very clear and interesting blog post. Thanks Ilya for taking the time to write it down.

I have a question about the learning rate schedule you recommend. For convex models trained with SGD, finding the optimal learning rate schedule is purely an optimization problem and therefore the optimal learning rate schedule should only depend on the training data and the loss function it-self. It is my understanding that there should be no need to use an held out validation set to find the optimal learning rate schedule in that case (leaving overfitting issues aside, assuming we have enough labeled samples to train on).

However for deep nets, you explicitly mentions that you need to decay learning rate based on the lack of improvement on the loss computed on some held-out validation set rather than using evolution of the cost of the training set.

What can go wrong when you do the learning rate scheduling based on the training cost instead of using an held-out validation set? Would the network just converge slower to an equivalent solution (assuming you still use the validation set for early stopping but not for the learning rate scheduling)? Or is it expected to converge to a significantly worse solution (e.g. by getting stuck on a plateau near a saddle point more easily)?

It seems that for deep networks, it might no longer be possible to separate the optimization problem from the learning / estimation problem. Do you have more intuitions to share on this topic?

1/15/2015 2:19 PM

Michael Klachko said...

Ilya, what do you think about capsules based neural networks of Geoff Hinton? Do you agree we need the additional structure, as opposed to "simple" layers of neurons?
Do you think we are gradually moving towards more biologically realistic models of information processing?

1/15/2015 9:51 PM

Ilya Sutskever said...

Olivier: basically, if you focus on the training data, you risk overfitting. If your training set is enormous in comparison to the size of the model, then overfitting is not a concern. But if your training set is smaller than the model, then as you keep training, eventually the validation (and hence test) error will start increasing. You definitely don't want to continue training once this happens.

However, it has been observed that a 2x-10x reduction in the LR size results in a very rapid reduction in both training and validation (and hence test) errors. Which is why, when you see that validation error no longer makes any progress with the large LR (so it may start increasing soon, which is bad), you reduce the LR, to get an additional gain.

I am pretty sure that this effect will hold true for convex problem as well -- this particular learning rate schedule attempts to find a parameter setting with the lowest validation error, which is something we care about much more than training error, which is relevant only to the extent it is correlated with the test error.

Optimization and generalization are intertwined, and that it is possible to optimize the training set better while doing worse on the test set.

Michael: it is possible that some types of additional structures will be helpful, especially if they are very general or easily trainable. As for biological realism, it's a matter of opinion. I think that our artificial neural networks have much in common to biological neural networks, but people who know more about real neurons may disagree.

1/17/2015 11:20 AM

Anonymous said...

You seem to equate 80s NNs with today's NNs. Beyond computers being slow, having less data, the NN tech has changed. Instead of punishing/readjusting each neuron, we now have layers of RBMs which do their own learning, seperately. This is an entirely different approach then what we had during 80s.

1/18/2015 6:29 AM

Anonymous said...

No one uses RBMs any more. All the state of the art models for speech and object recognition are the standard feedforward neural network, trained with backprop. The only algorithmic differences from 80s are ReLU and Dropout.

1/18/2015 2:17 PM

Yoshua Bengio said...

And initialization and depth.

1/18/2015 2:25 PM

Greg V said...

Hi Ilya, very nice post thanks.

You say:
> Make sure that you have a high-quality dataset of input-output examples that is large, representative, and has relatively clean labels. Learning is completely impossible without such a dataset.

My question is, do you think that humans really learn from this type of data? I find it implausible since it seems there are many things humans can learn from a handful of examples. On the other hand, I think I remember Yoshua arguing that, for vision, e.g., even if you can learn about some new object from a few examples, it is only because of the huge amount of "training data" we have processed in childhood.
Anyway, do you think there is some difference between the data humans can learn from, and what LDNNs can learn from?

1/18/2015 6:15 PM

Ilya Sutskever said...

It's obvious that humans learn from a very wide variety of information sources, and that input-output examples is only one of many such information sources. Humans get most of their "labels" indirectly, and only a small fraction of human learning is done with explicit input-output examples.

1/18/2015 10:24 PM

Anonymous said...

So is this post out of date?

http://deeplearning.net/tutorial/DBN.html

Or Kevin Murphy's latest book (he himself said it might be out of date, but) where he talks about stacked RBMs.

And I just saw this post

https://www.paypal-engineering.com/2015/01/12/deep-learning-on-hadoop-2-0-2/

where a data scientist at Paypal implemented deep learning on Hadoop, using a stack of RBMs.

Are you saying all these are behind the most recent state-of-art?

1/19/2015 2:58 AM

Yoshua Bengio said...

Stacks of unsupervised feature learning layers are STILL useful when you are in a regime with insufficient labeled examples, for transfer learning or domain adaptation. It is a regularizer. But when the number of labeled examples becomes large enough, the advantage of that regularizer becomes much less. I suspect however that this story is far from ended! There are other ways besides pre-training of combining supervised and unsupervised learning, and I believe that we still have a lot to improve in terms of our unsupervised learning algorithms.

1/19/2015 4:32 AM

Anonymous said...

Thanks for the response. There is a lot of terminology flying around on DL; You and LeCun are all about convnets, Hinton paper are mostly about RBMs, and they are considered "deep", utilizing backprop, in some shape of form. I guess I thought, naively, when there was a breakthrough on DL it was one specific method. It now looks like the breakthough is really about the net being deep (hence the name), in the network implementation a multitude of approaches can be utilized.

1/19/2015 4:50 AM

Yoshua Bengio said...

That is right. It is not about one specific algorithm, but about the architecture of the net being deep! There is even theory which only deals with the depth aspect, irrespective of how you learn. Of course everyone would like to have *the algorithm*, but we are not there yet!

1/19/2015 6:08 PM

Gary Bradski said...

Nice summary. One thing you've left out of the "ten step bio compute" speculation can be seen in robotics and in your dreams: ongoing (IMO causal-dynamic) state. The feedforward networks update a causal model of the world. Robots represent themselves in the world and use simulations in the model for planning and simulated learning. We do the same.

There is a lot more feedback than feedforward in the brain. Wires are expensive. The brain does this IMO to support this simulated world where perception actually takes place -- in the consensus of this abstracted physical model and it's deep NN inputs. So, 10 steps to feed data in, but really many more steps support the ongoing recognition.

Build a robot and you'll end up building some version of this interior causal model. I add dynamics because, unlike our NNs, our brain is mostly dynamically stable, not absolutely. Learning is never turned off, most of the stable patterns are there because the stability comes from the external world. Again, IMO. There are some key learning critical periods to bootstrap up the categorical structure that drives the internal model, again, IMHO.

Gary

1/24/2015 7:10 PM

Anonymous said...

How about the opinion that Deep Neural Networks are have no similiarities to biological neural networks and thus naming them Neural is deep offense to biology? Meaning that DNNs can be easily cheated, are not robust to clutter and noise, results of training are limited to specific data sets, etc.?

1/30/2015 1:05 AM

Edmund Ronald Ph. D. said...

Ysong - I've taken the liberty of linking to this post from my own Deep Neural blog.

I would like to thank Ilya for giving away all that sacred knowledge for free :)

Ilya, do you think recurrent nets will now be much more commonly used thanks to your work? I believe it was possible in the past to train them by reinforcement.

Edmund
http://deepneural.blogspot.fr/2015/03/go-and-read-ilya-sutskevers.html

3/12/2015 6:24 PM

Edmund Ronald Ph. D. said...

BTW, as "philosophical" comments from all comers seem to be allowed - I always think that the fact that these nets work just shows that we look at stuff that is adapted to our vision - in other words we have built alphabets for our hand, eyes,and quills; we have created roads and cars together, most of what we aim to recognize out there is actually already in some way evolved to be recognized. And so we should put stuff like astro data or even stock market data into a different category as it is not necessarily evolved to be understood.

Edmund

3/12/2015 6:33 PM

Anonymous said...

Gary Bradski: What the heck are you talking about?

4/11/2015 11:59 PM

Anonymous said...

so, Deep learning is essentially a self-created solution lookup table?

4/29/2015 2:38 AM

Free equity tips said...

wow...Great Blog ...So Much Things I have Learnt About deep Learning From This Blog....Thank You

11/26/2015 4:24 AM

Bjarke said...

Great post, thanks!

I have a question regarding preprocessing that I would love if you or Prof. Hinton could give me your thought on. You write the following:

"It is essential to center the data so that its mean is zero and so that the variance of each of its dimensions is one. Sometimes, when the input dimension varies by orders of magnitude, it is better to take the log(1 + x) of that dimension. Basically, it's important to find a faithful encoding of the input with zero mean and sensibly bounded dimensions. Doing so makes learning work much better."

This makes sense. The issue is that my dataset is highly sparse, meaning that it becomes difficult to obtain both unit variance and sensibly bounded dimensions. To obtain unit variance I must multiply by the std. deviation, which is roughly 0.25 for each channel, making around 30-50% of my values in each channel fall outside of what I would call a "sensibly bounded interval" such as [-3,3].

My dataset consists of roughly ~1.4 mio. observations of non-negative data and it is trained using a ConvNet with 8 layers with weights (conv. layers and dense layers). I am not modeling images or text (typical uses of ConvNets). One can imagine that 90% of my values are 0 and the rest are uniformly distributed on]0,10]]0,10].

I have the following questions:

1. Based on a knowledge of SGD (and perhaps ConvNets) - is it most important to aim for $\sigma=1\sigma=1$ or a sensible interval such as [3,3][3,3]?

2. What useful transformations could I do to fulfill both $\sigma=1\sigma=1$ and keeping my values in a sensible interval?

Note that I have asked the same question on http://stats.stackexchange.com/questions/188925/feature-standardization-for-convolutional-network-on-sparse-data in case you don't have time to reply. Feel very free to answer there instead.

1/01/2016 4:44 PM

jet essay - same day essay writing said...

A very big advantage of it is that the model is trainable! Thanks for the article and useful information! It helped a lot to clarify the situation!

2/12/2016 3:07 AM

Rocky said...

Thanks for sharing such a nice information. One can also get the details of online gate books for ie at Online ICE GATE Institute.

3/21/2016 2:53 AM

Post a Comment

Links to this post

Create a Link

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

Simple template. Powered by Blogger.