Deep Learning’s Most Important Ideas - A Brief Historical Review

Denny Britz

2020-07-29

Deep Learning is an extremely fast-moving field and the huge number of research papers and ideas can be overwhelming. Even seasoned researchers have a hard time telling company PR from real breakthroughs. The goal of this post is to review those ideas that have stood the test of time, which is perhaps the only significance test one should rely on. These ideas, or improvements of them, have been used over and over again. They’re known to work.

If you were to start in Deep Learning today, understanding and implementing each of these techniques would give you an excellent foundation for understanding recent research and working on your own projects. It’s what I believe the best way to get started. Working through papers in historical order is also a useful exercise to understand where the current techniques come from and why they were invented in the first place. Put another way, I will try to present a minimal set of ideas that most of the basic knowledge necessary to understand modern Deep Learning research.

A rather unique thing about Deep Learning is that its application domains (Vision, Natural Language, Speech, RL, etc) share the majority of techniques. For example, someone who has worked in Deep Learning for Computer Vision his whole career could quickly be productive in NLP research. The specific network architectures may differ, but the concepts, approaches and code are mostly the same. I will try to present ideas from various fields, but there are a few caveats about this list:

- My goal is not to give in-depth explanations or code examples for these techniques. It’s not easily possible to summarize long complex papers into a single paragraph. Instead, I will give a brief overview of each technique, its historical context, and links to papers and implementations. If you want to learn something, I highly recommend trying to re-produce some of these paper results from scratch in raw PyTorch without using existing code bases or high-level libraries.
- The list is biased towards my own knowledge and the fields I am familiar with. There are many exciting subfields that I don’t have experience with. I will stick to what most people would consider the popular mainstream
domains of Vision, Natural Language, Speech, and Reinforcement Learning / Games.

- I will only discuss research that has official or semi-official open source implementations that are known to work well. Some research isn’t easily reproducible because it involves huge engineering challenges, for example DeepMind’s AlphaGo or OpenAI’s Dota 2 AI, so I won’t highlight it here.
- Some choices are arbitrary. Often, rather similar techniques are published at around the same time. The goal of this post is not be a comprehensive review, but to expose someone new to the field to a cross-section of ideas that cover a lot of ground. For example, there may be hundreds of GAN variations, but to understand the general concept of GANs, it really doesn’t matter which one you study.

2012 - Tackling ImageNet with AlexNet and Dropout

Papers

Implementations
- AlexNet in PyTorch
- AlexNet in TensorFlow

AlexNet is often considered the algorithm responsible for the recent boom in Deep Learning and Artificial Intelligence research. It is a Deep Convolutional Neural Network based on the earlier LeNet developed by Yann LeCun. AlexNet beat previous methods at classifying images from the ImageNet dataset by a significant margin through a combination of GPU power and algorithmic advances. It demonstrated that neural networks actually work! AlexNet was also one of the first times Dropout Hinton et al. (2012) was used, which has since become a crucial component for improving the generalization ability of all kinds of Deep Learning models.

The architecture used by AlexNet, a sequence of Convolutional layers, ReLU non-linearity, and max-pooling, became the accepted standard that future Computer Vision architectures would extend and built upon. These days, software libraries such as PyTorch are so powerful, and compared to more recent architectures AlexNet is so simple, that it can be implemented in only a few lines of code. Note that many implementations of AlexNet, such as those linked above, use
the slight variation of the network described in One weird trick for parallelizing convolutional neural networks Krizhevsky (2014).

2013 - Playing Atari with Deep Reinforcement Learning

Papers

Implementations
- DQN in PyTorch
- DQN in TensorFlow

Building on top of the recent breakthroughs in image recognition and GPUs, a team at DeepMind managed to train a network to play Atari Games from raw pixel inputs. What’s more, the same neural network architecture learned to play seven different games without being told any game-specific rules, demonstrating the generality of the approach.

Reinforcement Learning differs from Supervised Learning, such as image classification, in that an agent must learn maximize to the sum of rewards over multiple time steps, such as winning a game, instead of just predicting a label. Because the agent interacts directly with the environment and each action affects the next, the training data is not independent and identically distributed (iid), which makes the training of many Machine Learning models quite unstable. This was solved by using techniques such as experience replay Lin (1992).
While there was no obvious algorithmic innovation that made this work, the research cleverly combined existing techniques, convolutional neural networks trained on GPUs and experience replay, with a few data processing tricks to achieve impressive results that most people would not have expected. This gave people confidence in extending Deep Reinforcement Learning techniques to tackle even more complex tasks such as Go, Dota 2, Starcraft 2, and others.

Atari Games Bellemare et al. (2013) have since become a standard benchmark in Reinforcement Learning research. The initial approach only solved (beat human baselines on) seven games, but over the coming years advances built on top of these ideas would start beating humans on an ever increasing number of games. One particular game, Montezuma’s Revenge, was famous for requiring long-term planning and was considered to be among the most difficult to solve. It was only recently Badia et al. (2020) Ecoffet et al. (2020) that techniques managed to beat human baselines on all 57 games.

2014 - Encoder-Decoder Networks with Attention

Papers
- Sequence to Sequence Learning with Neural Networks Sutskever, Vinyals, and Le (2014)
- Neural Machine Translation by Jointly Learning to Align and Translate Bahdanau, Cho, and Bengio (2016)
Deep Learning’s most impressive results had largely been on vision-related tasks and was driven by Convolutional Neural Networks. While the NLP community had success with Language Modeling and Translation using LSTM networks Hochreiter and Schmidhuber (1997) and Encoder-Decoder architectures Sutskever, Vinyals, and Le (2014), it was not until the invention of the attention mechanism Bahdanau, Cho, and Bengio (2016) that things started to work spectacularly well.

When processing language, each token, which could be a character, a word, or something in between, is fed into a recurrent network, such as an LSTM, which maintains a kind of memory of previously processed inputs. In other words, a sentence is very similar to a time series with each token being a time step. These recurrent models often had difficulty dealing with dependencies over long time horizons. When they process a sequence, they would easily "forget" earlier inputs because their gradients needed to propagate through many time steps. Optimizing these models with gradient descent was hard.

The new attention mechanism helped alleviate the problem. It gave the network an option to adaptively "look back" at earlier time steps by introducing shortcut connections. These connections allowed the network to decide which inputs are important when producing a specific output. The canonical example is
When producing an output word, it typically maps to one or more specific input words.

2014 - Adam Optimizer

Papers


Implementations

- Implementing Adam in Python
- PyTorch Adam implementation
- TensorFlow Adam implementation

Neural networks are trained by minimizing a loss function, such as the average classification error, using an optimizer. The optimizer is responsible for figuring out how to adjust the parameters of the network to make it learn the objective. Most optimizers are based on variations of Stochastic Gradient Descent (SGD). However, many of these optimizers contain tunable parameters such as a learning rate themselves. Finding the right settings for a specific problem not only reduces training time, but can also lead to better results due to finding a better local minimum of the loss function.
Big research labs often ran expensive hyperparameter searches that came up with complex learning rate schedules to get the best out of simple but hyperparameter-sensitive optimizers such as SGD. When they beat existing benchmarks, it sometimes was a result of spending a lot of money to tune the optimizer. Such details often went unmentioned in published research papers. Researchers who did not have the same budget to optimize their optimizer were stuck with worse results.

The Adam optimizer proposed to use the first and second moments of the gradients to automatically adapt the learning rate. The result turned out to be quite robust and less sensitive to hyperparameter choices. In other words, Adam often just works and did not require the same extensive tuning as other optimizers Sivaprasad et al. (2020). While an extremely well-tuned SGD could still get slightly better results, Adam made research more accessible because if something didn’t work, you knew it was unlikely to be the fault of a badly tuned optimizer.

2014/2015 - Generative Adversarial Networks (GANs)

Papers

- Generative Adversarial Networks Goodfellow et al. (2014)
- Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks Radford, Metz, and Chintala (2016)

Implementations

- DCGAN in PyTorch
- DCGAN in TensorFlow

The goal of generative models, such as variational autoencoders, is to create realistic-looking data samples, like these images of people’s faces you’ve probably seen somewhere. Because they have to model the full data distribution (many pixels!), and not just classify cats or dogs as a discriminative model would, such models are often quite difficult to train. Generative Adversarial Networks, or GANs, are one such type of model.

The basic idea behind GANs is to train two networks in tandem - a generator and a discriminator. The generator’s goal is to produce samples that fool the discriminator, which is trained to distinguish between real and generated images. Over time, the discriminator will become better at recognizing fakes, but the generator will also become better at fooling the discriminator and thus produce ever more realistic-looking samples. The very first iteration of GANs produced blurry low-resolution images and was quite unstable to train. But over time, variation and improvements such as DCGAN Radford, Metz, and Chintala (2016), Wasserstein GAN Arjovsky, Chintala, and Bottou (2017), CycleGAN
Figure 2: Visualization of samples from the model. Rightmost column shows the nearest training example of the neighboring sample, in order to demonstrate that the model has not memorized the training set. Samples are fair random draws, not cherry-picked. Unlike most other visualizations of deep generative models, these images show actual samples from the model distributions, not conditional means given samples of hidden units. Moreover, these samples are uncorrelated because the sampling process does not depend on Markov chain mixing. a) MNIST b) TFD c) CIFAR-10 (fully connected model) d) CIFAR-10 (convolutional discriminator and “deconvolutional” generator)

Figure 5: Source: https://developers.google.com/machine-learning/gan/gan_structure
Zhu et al. (2018), StyleGAN (v2) Karras et al. (2020), and many others have built upon this idea to produce high-resolution photorealistic images and videos.

2015 - Residual Networks (ResNet)

Papers
- Deep Residual Learning for Image Recognition He et al. (2015)

Implementations
- ResNet in PyTorch
- ResNet in Tensorflow

Researchers had been building on top of the AlexNet breakthrough for a while, inventing better-performing architectures based on Convolutional Neural Networks such as VGGNet Simonyan and Zisserman (2015), Inception Szegedy et al. (2014), and many others. ResNet was the next iteration in this rapid series of advances. To this day, ResNet variations are commonly used as a baseline model architecture for all kinds of tasks and as building blocks for more complex architectures.

What made ResNet special, apart from it winning first place in the ILSVRC 2015 classification challenge, was its depth compared to other network architectures. The deepest network presented in the paper had 1,000 layers and still performed well, though slightly worse than its 101 and 152 layer counterparts on the
benchmark tasks. Training such deep networks was a challenging optimization problem due to the vanishing gradients, which also appeared in sequence models. Not many researchers believed that training such extremely deep networks could lead to good stable results.

ResNet used identity shortcut connections that help the gradient flow. One way to interpret these connections is that ResNet only needs to learn 'deltas' from one layer to another, which is often easier than learning full transformations. Such identity connections were a special case of the connections proposed in Highway Networks Srivastava, Greff, and Schmidhuber (2015), which in turn were inspired by the gating mechanisms LSTMs used.

2017 - Transformers

Papers
- Attention is All You Need Vaswani et al. (2017)

Implementations
- PyTorch: Sequence-to-Sequence Modeling with nn.Transformer and Torch-Text
- Tensorflow: Transformer model for language understanding
- HuggingFace Transformers Library

Sequence-to-Sequence models with attention (described earlier in this post) worked quite well, but they had a few drawbacks due to their recurrent nature that required sequential computation. They were difficult to parallelize because they processed the input one step at a time. Each time step depends on the previous one. This also made it difficult to scale them to very long sequences. Even with their attention mechanism, they still struggled with modeling complex long-range dependencies. Most of the 'work' seemed to be done in the recurrent layers.

Transformers solved these issues by completely removing the recurrence and replacing it with multiple feed-forward self-attention layers, processing all inputs in parallel and producing relatively short (= easy to optimize with gradient descent) paths between inputs and outputs. This made them really fast to train, easy to scale, and able to process a lot more data. To tell the network about the order of the inputs, which was implicit in the recurrent model, Transformers used positional encodings Gehring et al. (2017). To learn more about how exactly transformers work, which can be a bit confusing at first, I recommend this illustrated guide.

To say that Transformers worked better than almost anyone expected would be an understatement. Over the next few years, they would become the standard architecture for the vast majority of NLP and other sequence tasks, and even make their way into architectures for computer vision.
Figure 1: The Transformer - model architecture.

Figure 6: Source: https://arxiv.org/abs/1706.03762
Pre-training refers to training a model to perform some task and then using the learned parameters as an initialization to learn a related task. This makes intuitive sense - a model that has learned to classify images as cats or dogs should have learned something general about images and furry animals. When this model is fine-tuned to classify foxes, we would expect it to do better than a model that must learn from scratch. Similarly, a model that has learned to predict the next word in a sentence should have learned something general about human language patterns. We would expect it to be a good initialization for related tasks like translation or sentiment analysis.

Pre-training and fine-tuning had been used successfully in both Computer Vision and NLP, but while it had been standard in vision for a long time, making it work well in NLP seemed more challenging. Most state-of-the-art results still came from fully supervised models. With the advent of transformers, researchers finally started to make pre-training work, resulting in approaches such as ELMo Peters et al. (2018), ULMFiT Howard and Ruder (2018) and OpenAI’s GPT. BERT was the latest of such developments and many consider it to have started a new era of NLP research. Instead of being pre-trained on predicting the next word, as most other models, it was pre-trained on predicting masked (intentionally removed) words anywhere in the sentence, and whether two sentences are likely
to follow each other. Note that these tasks don’t require labeled data. It can be trained on any text, a whole lot of it! This pre-trained model, which likely has learned some general properties about language, can then be fine-tuned to solve supervised tasks, such as question answering or sentiment prediction. BERT performed incredibly well across a wide variety of tasks. Companies such as HuggingFace made it easy to download and fine-tune BERT-like models for any NLP task. Since then, BERT has been built upon by advances such as XLNet Yang et al. (2020) and RoBERTa Liu et al. (2019) and ALBERT Lan et al. (2020).

2019/2020 and beyond - BIG Language Models, Self-Supervised Learning?

The clearest trend throughout the history of Deep Learning is perhaps that of the bitter lesson. Algorithmic advances for better parallelization (= more data) and more model parameters win over “smarter techniques” again and again. This trend seems to continue well into 2020 where GPT-3, a huge 175 billion parameter language model by OpenAI, shows unexpectedly good generalization abilities despite its simple training objective and standard architecture.

Playing into the same trend are approaches such as contrastive self-supervised learning, e.g. SimCLR, that make better use of unlabeled data. As models become bigger and faster to train, techniques that can make efficient use of the huge set of unlabeled data on web, and learn general-purpose knowledge that which can transferred to other tasks, are becoming more valuable and widely adopted.

Honorary mentions

If you really need more papers to look at...

- Distributed Representations of Words and Phrases and their Compositionality (2013)
- Speech recognition with deep recurrent neural networks (2013)
- Very Deep Convolutional Networks for Large-Scale Image Recognition (2014)
- Rethinking the Inception Architecture for Computer Vision (2015)
- Mastering the game of Go with deep neural networks and tree search (2016)
- Neural Architecture Search with Reinforcement Learning (2017)
- Mask R-CNN (2017)
- Dota 2 with Large Scale Deep Reinforcement Learning (2017-2019)
• The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks (2018)
• EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks (2019)


Badia, Adrià Puigdomènech, Bilal Piot, Steven Kapturowski, Pablo Sprechmann,


“BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding.”


Simonyan, Karen, and Andrew Zisserman. 2015. “Very Deep Convolutional


