Machine Learning Links and Lessons Learned

List of all the lessons learned, best practices, and links from my time studying machine learning.

Data and Features

- Normalizing inputs isn't guaranteed to help.
  - A lot of people say you should always normalize, but TBH in practice, it doesn't help for me all the time. This advice is very very dataset dependent, and it's up to you to figure out if normalizing will be helpful for your particular task.
  - In order to determine whether it will help or not, visualize your data to see what type of features you're dealing with. Do you have some features that have values around 1000 or 2000? Do you have some that are only 0.1 or 0.2? If so, normalizing those inputs is likely a good idea.
- The process in which you convert categorical features into numerical ones is very important.
  - You have to decide whether you want to use dummies variables, use bins, etc.
- Many different methods of normalization
  - Min-max normalization where you do (x - a)/(b-a) where x is the data point, a is the min value and b is the max value.
  - Max normalization where you do x/b where b is the max value.
  - L1 normalization where you do x/c where c is the summation of all the values.
  - L2 normalization where you do x/c where c is the square root of the summation of all the squared values.
  - Z score normalization where you do (x - d)/e where d is the mean and e is the standard deviation.

- Something interesting you can do if you either have little data or missing data is that you can use a neural network to predict those values based on the data that you currently have.
  - I believe the matrix imputation is the correct term.
- Data augmentation comes in many different forms. Noise injection is the most general form. While working with images, the number of options increases a whole lot.
  - Translations
  - Rotations
  - Brightness
  - Shears
- Whenever thinking about whether or not to even apply machine learning to your problem, always consider the type of data that you have to solve the problem. Let's consider a model where you want to predict whether someone will develop cancer in the next few months based on their health data. The data is your input into the model and the output is a binary result. Seems like a good place where we can apply machine learning right, given that we have a number of patients' data and cancer results? However, we should also be thinking about whether the health data that we have about the individual is *enough* to be able to make some conclusion about the patient. Does the health data accurately encompass the different amounts of variation there are from person to person? Are we accurately representing each input example with the most comprehensive amount of information? Because if the data doesn't have enough information/features, your model will risk just overfitting to noise or the training set, and won't really learn anything useful at all.
- Never underestimate the power of simply increasing the size of your training dataset when you're attempting to improve your model accuracy. If you have $100 for your machine learning project, spend like 90 of those dollars on good data collection and preprocessing.

Models

- For regression problems where the goal is to output some real valued number, linear regression should always be your first choice. It gets great accuracy on a lot of datasets. Obviously, this is very dependent on the type of data you have, but you should always first try linreg, and then move on to other more complex models.
- XGBoost has been one of the best models I've seen for a lot of regression or classification tasks.
  - Averaging multiple runs of XGBoost with different seeds helps to reduce model variance.
  - Model hyperparameter tuning is important for XGBoost
- Whenever your features are highly correlated with each other, running OLS will have a large variance, and therefore you should use ridge regression instead.
- KNN does poorly with high dimensional data because the distance metrics would get all messed up with the scales.
- One of the themes I hear over and over again is that the effective capacity of your model should match the complexity of your task.
  - For simple house price prediction, a linear regression model might do.
  - For object segmentation, a large and specialized CNN will be a necessity.
- Use RNNs for time series data and natural language data since NLP data is contains dependencies

between different steps in the input.

Hyperparameters

- Optimizing hyperparameters can be a bit of a pain and can be really computationally expensive, but sometimes they can make anywhere from a 1 - 5 percent improvement. In some use cases, that can be the difference between a good product and a great product.
- Not saying that everyone should invest in a Titan X GPU to do large grid searches for optimizing hyperparameters, but it's very important to develop a structured way for how your going to go about it.
  - The easiest and simplest way is to just write a nested for loop that goes through different values, and keeps track of the models that perform the best.
  - You can also use specialized libraries and functions that will make the code look a little nicer.
    - Scikit-Learn has a good one
- SVM's
  - What are C and Gamma?
- The Deep Learning book says that the learning rate is the most important hyperparameter in most deep networks, and I do have to agree. A low LR makes training unbearably slow (and could get stuck in some minima), while a high LR can make training extremely unstable.

Tensorflow/Deep Nets

- Always make sure your model is able to overfit to a couple pieces of the training data. Once you know that the network is learning correctly and is able to overfit, then you can work on building a more general model.
- Evaluation of your Tensorflow graph only happens at runtime. Your job is to first define the variables and the placeholders and the operations (which together make up the graph) and then you can execute this graph in a session (execution environment), where you feed in values into the placeholders.
- Regularization is huuuuugely important with deep nets.
  - Weight Decay
  - Dropout
    - Seriously. Use it. It's sooo simple, but it's so good at helping to prevent overfitting and thus gives your test accuracy a nice boost most of the time.
  - Data Augmentation
  - Early Stopping
    - Really effective. Basically, just stop training when the validation error stops decreasing. Because if you training for any longer, you're just going to overfit to the training dataset more, and that's not going to help you with that test set accuracy. And honestly, this method of regularization is just really easy to implement.
  - Batch normalization
  - Label smoothing
  - Model averaging
  - Regularizers are thought to helpful the generalization ability of networks, but are not the *main*

reason that deep nets generalize so well. The idea is that you don't use all of the above, but if you are having overfitting issues, you should use some of these to combat those issues.
- That being said, modal architecture and the structure of your layers is sometimes thought to be more important.
- Always always always do controlled experiments. By this, I mean only change one variable at a time when you're trying to tune your model. There is an incredible number of hyperparameters and design decisions that you can change. Here are a few examples.
  - Network architecture
    - Number of convolutional layers in a CNN
    - Number of LSTM units in an RNN
  - Choice of optimizer
  - Learning rate
  - Regularization
  - Data augmentation
  - Data preprocessing
  - Batch size If you try to change too many of the above variables at once, you'll lose track of which changes really had the most impact. Once you make changes to the variables, always keep track of what impact that had on the overall accuracy or whatever other metric you're using.

Deep Learning Frameworks

- Keras - My friend and I have a joke where we say that you'll have a greater number of lines in your code just doing Keras imports, compared to the actual code because the functions are so incredibly high level. Like seriously. You could load a pretrained network and finetune it on your own task in like 6 lines. It's incredible. This is definitely the framework I use for hackathons and when I'm in a time crunch, but I think if you really really want to learn ML and DL, relying on Keras's nice API might not be the best call.
- Tensorflow - This is my go-to deep library nowdays. Honestly I think it has the steepest learning curve because it takes quite a while to get comfortable with the ideas of Tensorflow variables, placeholders, and building/executing graphs. One of the big plus sides to Tensorflow is the number of Github and Stackoverflow help you can get. You can find the answer to almost any error you get in Tensorflow because someone has likely run into it before. I think that's hugely helpful.
- Torch - 2015 was definitely the year of Torch, but unless you really want to learn Lua, PyTorch is probably the way to go now. However, there's a lot of good documentation and tutorials associated with Torch, so that's a good upside.
- PyTorch - My other friend and I have this joke where we say that if you're running into a bug in PyTorch, you could probably read the entirety of PyTorch's documentation in less than 2 hours and you still wouldn't find your answer LOL. But honestly, so many AI researchers have been raving about it, so it's definitely worth giving it a shot even though it's still pretty young. I think Tensorflow and PyTorch will be the 2 frameworks that will start to take over the DL framework space.

- Caffe and Caffe2 - Never played around with Caffe, but this was one of the first deep learning libraries out there. Caffe2 is notable because it's the production framework that Facebook uses to serve its models. According to Soumith Chintala, researchers at Facebook will try out new models and research ideas using PyTorch and will deploy using Caffe2.

CNNs

- Use CNNs for *any* image related task. It's really hard for me to think of any image processing task that hasn't been absoluted revolutionized by CNNs.
  - That being said, you might not want to use CNNs if you have latency, power, computation, or memory constraints. In a lot of different areas (IoT for example), some of the downsides of using a CNN flare up.
- Transfer learning is harder than it looks. Found out from firsthand experience. During a hackathon, my friend and I wanted to determine whether someone has bad posture or not (from a picture) and so we spent quite a bit of time creating a 500 image dataset, but even with using a pretrained model, chopping off the last layer, and retraining it, and while the network was able to learn and get a decent accuracy on the training set, the validation and testing accuracies weren't up to par signaling that overfitting might be a problem (due to our small dataset). Moral of the story is don't think that transfer learning will come and save your image processing task. Take a good amount of time to create a solid dataset, and understand what type of model you'll need, and what kind of modifications you'll need to make to the pretrained network.
  - Transfer learning is also interesting because there are two different ways that you can go with it. You can use transfer learning for finetuning. This is where you take a pretrained CNN (trained on Imagenet), chop off the last fully connected layer, add an FC layer specific to your task, and then retrain on your dataset. However, there's also another interesting approach called transfer learning for feature extraction. This is where you take a pretrained CNN, pass your data through the CNN, get the output activations from the last convolutional layer, and then use that feature representation as your data for training a more simple model like an SVM or linear regression.

NLP

- Use pretrained word embeddings whenever you can. From my experience, it's just a lot less hassle, and quite frankly I'm not sure if you even get a performance improvement if you try to train them jointly with whatever other main task you want to solve. It's task-dependent I guess. For something simple like sentiment analysis though, pretrained word vectors worked perfectly.
  - Glove word embeddings work great for me.

ML Project Advice

- Create your machine learning pipeline first and then start to worry about how you can tune and make your model better later.
  - For example, if you're creating an image classification model, make sure you're able to load data into your program, create test/train matrices, create a very simple model (using one of the

DL libraries), create your training loop, and make sure the network is learning something and that you can get to some baseline accuracy. Only once you've done these steps can you start to worry about things like regularization, data augmentation, etc.
    - Too many times you can get over complicated with the model and hyperparameters when your issues may lie just in the way you're loading in data or creating your training batches. Be sure to get those simple parts of the machine learning pipeline down.
    - Another benefit to making sure you have a minimal but working end to end pipeline is that you're able to track performance metrics as you start to change your model and tune your hyperparameters.
- If you're trying to create some sort of end product, 80% of your ML project will honestly be just doing front-end and back-end work. And even within that 20% of ML work, a lot of it will probably be dataset creation or preprocessing.
- Always divide your data into train and validation (and test if you want) sets. Checking performance on your validation set at certain points during training will help you determine whether the network is learning and when overfitting starts to happen.
- Always shuffle your data when you're creating training batches.

Kaggle

- Bit of a love/hate relationship with Kaggle. I think it's great for beginners in machine learning who are looking to get more practical experience. I can't tell you how much it helped me to go through the process of loading in data with Pandas, creating a model with Tensorflow/Scikit-Learn, training the model, and fine tuning to get good performance. Seeing how your model stacks up against the competition afterwards is a cool feeling as well.
- The part of Kaggle that I don't really enjoy is how much feature engineering is required to really get into the top 10-15% of the leaderboard. You have to be really committed to data visualization and hyperparameter tuning. Some may argue that this is a crucial part of being a data scientist/machine learning engineer, and I do agree that it's important in real world problem spaces. I think the main point here is that if you want to get good at Kaggle competitions, there's no easy road, and you'll have to do a lot of practice (which is not bad!).
- If you're looking to get better at Kaggle competitions, I'd recommend reading their blog. They interview a lot of the competition winners, which will give you a good look at the type of ML models and pipelines needed to get good performance.

Debugging ML Models

- What should you do when your ML model's accuracy just isn't good enough?
- If your train accuracy is great but your validation accuracy is poor, this likely means that your model has overfit to the training data.
    - Reduce model complexity
    - Gather more data
        - Data augmentation helps (at least for image data)
        - Honestly, just investing more time/money in this component of your ML model is almost always well worth it.

    - Regularization
- If your training accuracy just isn't good enough, you could be suffering from underfitting.
    - Increase model complexity
        - More layers or more units in each layer
    - Don't try to add more data. Your model clearly isn't able to classify the examples in the current dataset, so adding more data won't help.
- Any other problem you might be having.
    - Check the quality of your training data and make sure you're loading everything in properly
    - Adjust hyperparameters
        - I know it's tedious and it takes a boatload of time, but it can *sometimes* be worth.
        - Grid search might be the best approach, but keep in mind it can get extremely computationally expensive.
        - I heard random search is also surprisingly effective, but haven't tried it myself.
    - Restart with a very simple model and only a couple of training points, and make sure your model is able to learn that data. Once it gets 100% accuracy, start increasing the complexity of the model, as well as loading in more and more of your whole dataset.

Best Courses

- Stanford CS 231N - CNNs
- Stanford CS 224D - Deep Learning for NLP
- David Silver's Reinforcement Learning Course
- Andrew Ng Machine Learning Course
- Stanford CS 229 - Pretty much the same as the Coursera course
- UC Berkeley Kaggle Decal
- Short MIT Intro to DL Course
- Udacity Deep Learning
- Deep Learning School Montreal 2016

Best Python Libraries for ML

- Numpy
- Scikit-Learn
- Matplotlib
- Pandas

Most Important Deep Learning Papers

Not a comprehensive list by any sense. I just thought these papers were incredibly influential in getting deep learning to where it is today.

- AlexNet
- GoogLeNet
- VGGNet

- ZFNet
- ResNet
- R-CNN
- Fast R-CNN
- Adversarial Images
- Generative Adversarial Networks
- Spatial Transformer Networks
- DCGAN
- Synthetic Gradients
- Memory Networks
- Mixture of Experts
- Neural Turing Machines
- Alpha Go
- Atari DQN
- Word2Vec
- GloVe
- A3C
- Gradient Descent by Gradient Descent
- Rethinking Generalization
- Densely Connected CNNs
- EBGAN
- Wasserstein GAN
- Style Transfer
- Pixel RNN
- Dynamic Coattention Networks
- Convolutional Seq2Seq Learning
- Seq2Seq
- Dropout
- Batch Norm
- Large Batch Training
- Transfer Learning
- Adam
- Speech Recognition
- Relational Networks

Other Interesting Links

- Neural Network Playground
- ConvNetJS Demo
- Stanford Sentiment Analysis Demo
- Deep Learning's Relation to Physics
- Interpretability of AI
- Picking up Deep Learning as an Engineer

Best Blogs

- Andrej Karpathy
- Google Research
- Neil Lawrence
- Qure.ai
- Brandon Amos
- Denny Britz
- Moritz Hardt
- Deepmind
- Machine Learning Mastery
- Smerity
- The Neural Perspective
- Pete Warden
- Kevin Zakka
- Thomas Dinsmore
- Arthur Juliani
- CleverHans
- Off the Convex Path
- Berkeley AI Research
- Facebook AI Research
- Salesforce Research
- OpenAI
- Lab41
- Distill
- My blog :)

Random Thoughts

- It's interesting and worrying that the process in which neural nets and deep learning work is still such a black box. If an algorithm comes to the conclusion that a number is let's say classified as a 7, we don't really know how the algorithm came to that result because it's not hardcoded anywhere. It's hidden behind millions of gradients and derivatives. So if we wanted to use deep learning in a more important field like medicine, the doctor who is using this technology should be able to know why the algorithm is giving the result it's giving. Doctors also generally don't like the idea of probabilistic results.
    - "People still don't trust black-boxes. They need to see the audit trails on how decisions were made, something most AI technologies can't furnish."
    - Medical Diagnosis, Insurance claim rejections, and loan/mortgage applications.
        - You can't just have a ML system that basically outputs a binary response for whether or not someone has a certain disease without being able to point to some feature in the input data that explains the corresponding output.
        - Yoshua Bengio, though, argues that all you need is statistical reassurance about the general

ability of the trained model.
    - Jeff Dean mentioned that this interpretability problem is an issue in some domains and not an issue with others. I think that's the way you have to look at this topic, it's should be a case-by-case basis and we shouldn't really make broad statements saying lack of interpretability is terrible or lack of interpretability is something we can deal with.
- If you have problems where you can generate a lot of synthetic data, for example with reinforcement learning with arcade games. You can simulate so many rounds of games and get so much data off of that for pretty much nothing.
- Important to differentiate between which problems just need more data and compute to work better and which problems really need core algorithmic breakthroughs to really work.
- Breakthroughs over the past 5 or so years have mostly happened because of improvements in data/compute/infrastructure, not necessarily improvements in the algorithms themselves. At some point, we're going to have to make those better. Compute and data isn't going to be enough at some point.
- How is there is a trust between the human and the machine. In the past, machines would work exactly the way it is supposed to (as long as servers don't go down), This is kind of a new frontier where you don't really know that given a certain input, what the output will look like.
- The main reasons that deep learning methods have really worked over the last few years are:
    - The public availability of large datasets. Aka Thank you Fei-Fei Li!
    - Compute power. Aka Thanks Nvidia
    - Interesting models that take advantage of certain characteristics in data.
        - CNNs using convolutional filters to analyze images.
        - RNNs using recurrent networks to exploit sequence dependence in natural language data.

Research Ideas

- Creating systems that don't need as much training data to be effective.
    - One shot and zero shot learning.
- Networks that have specialized memory units that hold specific past data points and their labels.
    - Neural Turing Machine and Differentiable Neural Computer have external memories.
- Determining the actually useful things that GANs can do.
    - Generating synthetic data
    - Usage as a feature extractor to understand more about your data's distribution in an unsupervised learning setting (specifically one where you have a lot of unlabeled data), and then using those features for a different supervised learning task.
- Creating models with less labeled training data using one shot learning, learning from unstructured data, weak labels,
- Making reinforcement learning algorithms better in environments with sparse reward signals.
- How to make computations faster on hardware? How to make memory storage and access faster? How does everything change when you try distributing the workload to lots of different servers?
- Multitask learning in order to have single networks that able to solve a bunch of different tasks given input, rather than having one (CNN) for image inputs and one (RNN) for language inputs.
    - The main thing here is that we'd like to be able to compute some shared representation of the

input so that we're able to do some sort of analysis of it, regardless of whether the input is an image or text or speech, etc.
- Adding more variety to your loss function. A traditional loss function just represents the difference between a network's prediction and the true label, and our optimization procedure seeks to minimize that. We can also try to get creative with our loss functions and add soft constraints on the values of the weights (weight decay) or the values of the activations, or honestly whatever desirable property we want in our model.

Other

- Jupyter Notebook is the **best**. Seriously. It's so nice for ML because visualizing data and understanding dimensionality of your variables is so vital, and having an environment where you can do both of those things easily is very helpful.
- The number of clusters that you choose in K nearest neighbors is a hidden latent variable because it is something that isn't observed in the actual dataset.
- We use dimensionality reduction because data may appear high dimensional but there may only be a small number of degrees of variability.
- Don't bother with unsupervised learning unless you have a really simple task and you want to use K-Means, or if you're using PCA to reduce the dimensionality of data, or if you're playing around with autoencoders.. PCA can be ridiculously useful sometimes.
- Model compression is something that becomes extremely important when you're trying to deploy pretrained models onto mobile devices. Training a model offline with GPUs and then making sure that the network size is small enough so that end users can just run inference on those models at test time. Most of the time, the state of the art CNN models (like Inception or ResNet) are so huge that all the parameters can't possibly fit on one device. That's where model compression comes in.
- Haven't played around a whole lot with auto-encoders, but they seem really interesting. The basic idea is that you want a network that maps your input into another representation h, and then from h back to your original input. This is interesting because throughout the process of training, the weight values in between your x and h representations could give us a lot of good information about the type of data distribution we have in our training set. A really cool unsupervised learning method.