Deep Learning for Beginners

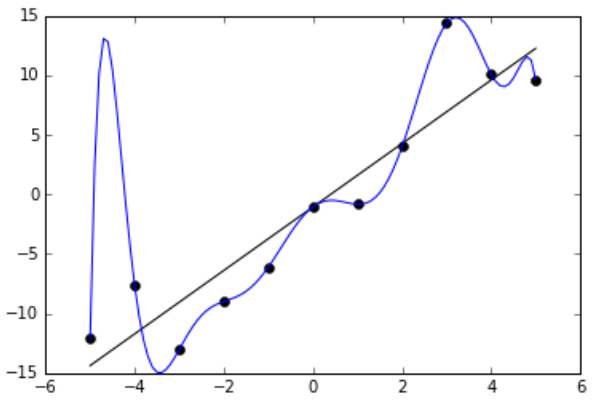
Deep Learning for Beginners

Notes for "Deep Learning" by Ian Goodfellow, Yoshua Bengio, and Aaron Courville.

Machine Learning

- Machine learning is a branch of statistics that uses samples to approximate functions.
  - We have a true underlying function or distribution that generates data, but we don't know what it is.
  - We can sample this function, and these samples form our training data.
- Example image captioning:
  - Function: $f^\star(\text{image}) \rightarrow \text{description}$.
  - Samples: $\text{data} \in (\text{image}, \text{description})$.
  - Note: since there are many valid descriptions, the description is a distribution in text space: $\text{description} \sim \text{Text}$.
- The goal of machine is to find models that:
  - Have enough representation power to closely approximate the true function.
  - Have an efficient algorithm that uses training data to find good approximations of the function.
  - And the approximation must generalize to return good outputs for unseen inputs.
- Possible applications of machine learning:
  - Convert inputs into another form - learn "information", extract it and express it. eg: image classification, image captioning.
  - Predict the missing or future values of a sequence - learn "causality", and predict it.
  - Synthesise similar outputs - learn "structure", and generate it.

Generalization and Overfitting.



- Overfitting is when you find a good model of the training data, but this model doesn't generalize.
  - For example: a student who has memorized the answers to training tests will score well on a training test, but might scores badly on the final test.
- There are several tradeoffs:

    - Model representation capacity: a weak model cannot model the function but a powerful model is more prone to overfitting.
    - Training iterations: training too little doesn't give enough time to fit the function, training too much gives more time to overfit.
    - You need to find a middle ground between a weak model and an overfitted model.
- The standard technique is to do cross validation:
    - Set aside "test data" which is never trained upon.
    - After all training is complete, we run the model on the final test data.
    - You cannot tweak the model after the final test (of course you can gather more data).
    - If training the model happens in stages, you need to withhold test data for each stage.
- Deep learning is one branch of machine learning techniques. It is a powerful model that has also been successful at generalizing.

Feedforward Networks

Feedforward networks represents $y = f^\star(x)$ with a function family:

$$u = f(x;\theta)$$

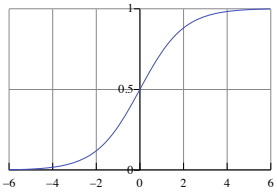
- θ are the model parameters. This could be thousands or millions of parameters $\theta_1 \dots \theta_T$.
- f is a family of functions. $f(x;\theta)$ is a single function of x. u is the output of the model.
- You can imagine if you chose a sufficiently general family of functions, chances are, one of them will resemble $f^\star$.
- For example: let the parameters represent a matrix and a vector: $f(\vec{x};\theta)() = \begin{bmatrix}\theta_0 & \theta_1 \\ \theta_2 & \theta_3\end{bmatrix}\vec{x} + \begin{bmatrix}\theta_4 \\ \theta_5\end{bmatrix}$

Designing the Output Layer.

The most common output layer is:

$$f(x;M,b) = g(Mx+b)$$

- The parameters in θ are used as M and b.
- The linear part $Mx + b$ ensures that your output depends on all inputs.
- The nonlinear part $g(x)$ allows you to fit the distributon of y.
- For example for input of photos, the output distribution could be:
    - Linear: $y \in \mathbb{R}$. eg cuteness of the photo
    - Sigmoid: $y \in [0,1]$. eg probability its a cat
    - Softmax: $y \in \mathbb{R}^C$ and $\sum y = 1$. eg. probability its one of C breeds of cats
- To ensure $g(x)$ fits the distribution, you can use:
    - Linear: $g(x) = x$.
    - Sigmoid: $g(x) = \frac{1}{1+e^{-x}}$.



  - Softmax: $g(x)_c = \frac{e^{x_c}}{\sum_i e^{x_i}}$.
- Softmax is actually under-constrained, and often x_0 is set to 1. In this case sigmoid is just softmax in 2 variables.
- There is theory behind why these g's are good choices, but there are many different choices.

Finding θ

Find θ by solving the following optimization problem for J the cost function:

$$\min_{\theta \in \text{models}} J\Big(y, f(x;\theta)\Big)$$

- Deep learning is successful because there is a good family of algorithms to calculate $\min$.
- That algorithms are all variations of gradient descent:

```
theta = initial_random_values()
loop {
  xs = fetch_inputs()
  ys = fetch_outputs()
  us = model(theta)(xs)
  cost = J(ys, us)
  if cost < threshold: exit;
  theta = theta - gradient(cost)
}
```

- Intuitively, at every θ you chose the direction that reduces the cost the most.
- This requires you to compute the gradient $\frac{d\text{cost}}{d\theta_t}$.
- You don't want the gradient to be near 0 because you learn too slowly or near $\inf$ because it is not stable.
- This is a greedy algorithm, and thus might converge but into a local minimum.

Choosing the Cost Function

- This cost function could be anything:
  - Sum of absolute errors: $J = \sum |y - u|$.
  - Sum of square errors: $J = \sum (y - u)^2$.
  - As long as the minimum occurs when the distributions are the same, in theory it would work.
- One good idea is that u represents the parameters of the distribution of y.
  - Rationale: often natural processes are fuzzy, and any input might have a range of outputs.
  - This approach also gives a smooth measure of how accurate we are.
  - The maximum likelihood principle says that: $\theta_{\text{ML}} = \arg\max_\theta p(y;u)$

    - Thus we want to minimize: $J = -p(y; u)$
    - For i samples: $J = -\prod_i p(y_i; u)$
    - Taking log both sides: $J' = -\sum_i \log p(y_i; u)$.
    - This is called cross-entropy.
- Applying the idea for: $y \sim \text{Gaussian}(\text{center} = u)$:
    - $p(y; u) = e^{-(y-u)^2}$.
    - $J = -\sum \log e^{-(y-u)^2} = \sum (y-u)^2$
    - This motivates sum of squares as a good choice.

Regularization

- Regularization techniques are methods that attempt to reduce generalization error.
    - It is not meant to improve the training error.
- Prefer smaller θ values:
    - By adding some function of θ into J we can encourage small parameters.
    - L^2: $J' = J + \sum |\theta|^2$
    - L^1: $J' = J + \sum |\theta|$
    - L^0 is not smooth.
    - Note for $\theta \to Mx + b$ usually only M is added.
- Data augmentation:
    - Having more examples reduces overfitting.
    - Also consider generating valid new data from existing data.
    - Rotation, stretch existing images to make new images.
    - Injecting small noise into x, into layers, into parameters.
- Multi-Task learning:
    - Share a layer between several different tasks.
    - The layer is forced to choose useful features that is relevant to a general set of tasks.
- Early stopping:
    - Keep a test data set, called the validation set, that is never trained on
    - Stop training when the cost on the validation set stops decreasing.
    - You need an extra test set to truly judge the the final.
- Parameter sharing:
    - If you know invariants about your data, encode that into your parameter choice.
    - For example: images are translationally invariant, so each small patch should have the same parameters.
- Dropout:
    - Randomly turn off some neurons in the layer.
    - Neurons learn to not take input data for granted.
- Adversarial:
    - Try to make the points near training points constant, by generating adversarial data near these points.

Deep Feedforward Networks

Deep feedforward networks instead use:

$$u = f(x, \theta) = f^N(\dots f^1(x; \theta^1) \dots; \theta^N)$$

- This model has N layers.
  - $f^1 \dots f^{N-1}$: hidden layers.
  - f^N: output layer.
- A deep model sounds like a bad idea because it needs more parameters.
- In practise, it actually needs fewer parameters, and the models perform better (why?).
- One possible reason is that each layer learns higher and higher level features of the data.
- Residual models: $f'^n(x) = f^n(x) + x^{n-1}$.
  - Data can come from the past, we add on some more details to it.

Designing Hidden Layers.

The most common hidden layer is:

$$f^n(x) = g(Mx + b)$$

- The hidden layers have the same structure as the output layer.
- However the $g(x)$ which work well for the output layer don't work well for the hidden layers.
- The simplest and most successful g is the rectified linear unit (ReLU): $g(x) = \max(0, x)$.
  - Compared to sigmoid, the gradients of ReLU does not approach zero when x is very big.
- Other common non-linear functions include:
- Modulated ReLU: $g(x) = \max(0, x) + \alpha \min(0, x)$.
  - Where alpha is -1, very small, or a model parameter itself.
  - The intuition is that this function has a slope for x < 0.
  - In practise there is no absolute winner between this and ReLU.
- Maxout: $g(x)_i = \max_{j \in G(i)} x_j$
  - G partitions the range $[1..I]$ into subsets $[1..m], [m+1..2m], [I-m..I]$.
  - For comparison ReLU is $\mathbb{R}^n \to \mathbb{R}^n$, and maxout is $\mathbb{R}^n \to \mathbb{R}^{\frac{n}{m}}$.
  - It is the max of each bundle of m inputs, think of it as m piecewise linear sections.
- Linear: $g(x) = x$
  - After multiplying with the next layer up, it is equivalent to: $f^n(x) = g'(NMx + b')$
  - It's useful because you can use it to narrow N and M, which has less parameters.

Optimizaton Methods

- The methods we use is based on stochastic gradient descent:
  - Choose a subset of the training data (a minibatch), and calculate the gradient from that.
  - Benefit: does not depend on training set size, but on minibatch size.
- There are many ways to do gradient descent (using: gradient g, learning rate ϵ, gradient update Δ)
  - Gradient descent - use gradient: $\Delta = \epsilon g$.
  - Momentum - use exponential decayed gradient: $\Delta = \epsilon \sum e^{-t} g_t$.
  - Adaptive learning rate where $\epsilon = \epsilon_t$:
    - AdaGrad - slow learning on gradient magnitude: $\epsilon_t = \frac{\epsilon}{\delta + \sqrt{\sum g_t^2}}$.
    - RMSProp - slow learning on exponentially decayed gradient magnitude: $\epsilon_t = \frac{\epsilon}{\sqrt{\delta + \sum e^{-t} g_t^2}}$.
    - Adam - complicated.
- Newton's method: it's hard to apply due to technical reasons.

- Batch normalization is a layer with the transform: $y = m\frac{x-\mu}{\sigma} + b$
  - m and b are learnable, while μ and σ are average and standard deviation.
  - This means that the layers can be fully independent (assume the distribution of the previous layer).
- Curriculum learning: provide easier things to learn first then mix harder things in.

Simplifying the Network

- At this point, we have enough basis to design and optimize deep networks.
- However, these models are very general and large.
  - If your network has N layers each with S inputs/outputs, the parameter space is $|\theta| = O(NS^2)$.
  - This has two downsides: overfitting, and longer training time.
- There are many methods to reduce parameter space:
  - Find symmetries in the problem and choose layers that are invariant about the symmetry.
  - Create layers with lower output dimensionality, the network must summarize information into a more compact representation.

Convolution Networks

A convolutional network simplifies some layers by using convolution instead of matrix multiply (denoted with a star):

$$f^n(x) = g(\theta^n * x)$$

- It is used for data that is spatially distributed, and works for 1d, 2d and 3d data.
  - 1d: $(\theta * x)_i = \sum_a \theta_a x_{i+a}$
  - 2d: $(\theta * x)_{ij} = \sum_a \sum_b \theta_{ab} x_{ab+ij}$
- It's slightly from the mathematical definition, but has the same idea: the output at each point is a weighted sum of nearby points.
- Benefits:
  - Captures the notion of locality, if θ is zero, except in a window w wide near $i = 0$.
  - Captures the notion of translation invariance, as the same θ are used for each point.
  - Reduces the number of model parameters from $O(S^2)$, to $O(w^2)$.
- If there are n layers of convolution, one base value will be able to influence the outputs in a wn radius.
- Practical considerations:
  - Pad the edges with 0, and how far to pad.
  - Tiled convolutions (you rotate between different convolutions).

Pooling

A common layer used in unison with convnets is max pooling:

$$f^n(x)_i = \max_{j \in G(i)} x_j$$

- It is the same structure as maxout, and equivalent in 1d.

- For higher dimensions G partitions the input space into tiles.
- This reduces the size of the input data, and can be considered as collapsing a local region of the inputs into a summary.
- It is also invariant to small translations.

Recurrent Networks

Recurrent networks use previous outputs as inputs, forming a recurrence:

$$s^{(t)} = f(s^{(t-1)}, x^{(t)}; \theta)$$

- The state s contains a summary of the past, while x is the inputs that arrive at each step.
- It is a simpler model than a fully dynamic: $s^{(t)} = d^{(t)}(x^{(t)}, \dots x^{(1)}; \theta')$
  - All the θ's are shared across time - the recurrent network assumes time invariance.
  - A RNN can learn for any input length, while a fully dynamic model needs a different g for each input length.
- Output: the model may return $y^{(t)}$ at each time step:
  - No output during steps, only the final state matters. Eg: sentiment analysis.
  - $y^{(t)} = s^{(t)}$, the model has no internal state and thus less powerful. But it is easier to train, since the training data y is just s.
  - $y^{(t)} = o(s^{(t)})$, use an output layer to transform (and hide) the internal state. But training is more indirect and harder.
  - As always we prefer to think of y as the parameters to a distribution.
- The output chosen may be fed back to f as extra inputs. If not fed in, the y are conditionally independent of each other.
  - When generating a sentence, we need conditional dependence between words, eg: A-A and B-B might be valid, but A-B might be invalid.
- Completion:
  - Finish when input ends. This works for $x^{(t)} \to y^{(t)}$.
  - Extra output $y^{(t)}_{\text{end}}$ with the probability the output has completed.
  - Extra output $y^{(t)}_{\text{length}}$ with the length of output remaining/total.
- Optimization is done using the same gradient descent class of methods.
  - Gradients are calculated by expanding the recurrence to a flat formula, called back-propagation through time (BPTT).
- One difficult aspect of BPTT is the gradient $\Delta = \frac{\partial}{\partial s^t}$:
  - $\Delta > 0$: the state explodes, and provide unstable gradient. The solution is to clip the gradient updates to a reasonable size during descent.
  - $\Delta \approx 0$: this allows the state to persist for a long time, howevr the gradient descent method needs a gradient to work.
  - $\Delta < 0$: the RNN is in a constant state of information loss.
- There are variants of RNN that impose a simple prior to help preserve state $s^{(t)} \approx s^{(t-1)}$:
  - $s^{(t)} = f_t s^{(t-1)} + f(\dots)$: we get a direct first derivative $\frac{\partial}{\partial x}$
  - It lets us pass along a gradient from previous steps, even when f itself has zero gradient.
- Long short-term memory (LSTM) model input, output and forgetting:
  $s^{(t)} = f_t s^{(t-1)} + i_t f(s^{(t-1)}, x^{(t)}; \theta)$
  - The ouput is: $y^{(t)} = o_t s^{(t)}$
  - It uses probabilities (known as gates): o_t output, f_t forgetting, i_t input.

    - The gates are usually a sigmoid layer: $o_t = g(Mx + b) = \frac{1}{1+e^{Mx+b}}$.
    - Long term information is preserved, because generating new data g and using it i are decoupled.
    - Gradients are preserved more as there is a direct connection between the past and future.
- Gated recurrent unit (GRU) are a simpler model: $s^{(t)} = (1 - u_t)s^{(t-1)} + u_t f(r_t s^{(t-1)}, x^{(t)}; \theta)$
    - The gates: u_t update, r_t reset.
    - There is no clear winner.
- For dropout, prefer $d(f(s, x; \theta))$ not storing information, over $d(s)$ losing information.
- Memory Networks and attention mechanisms.

Useful Data Sets

- There are broad categories of input data, the applications are limitless.
- Images vector $[0 - 1]^{WH}$: image to label, image to description.
- Audio vector for each time slice: speech to text.
- Text embed each word into vector $[0 - 1]^N$: translation.
- Knowledge Graphs: question answering.

Autoencoders

An autoencoder has two functions, which encode f and decode g from input space to representation space. The objective is:

$$J = L(x, g(f(x)))$$

- L is the loss function, and is low when images are similar.
- The idea is that the representation space learns important features.
- To prevent overfitting we have some additional regularization tools:
    - Sparse autoencoders minimize: $J' = J + S(f(x))$. This is a regularizer on representation space.
    - Denoising autoencoders minimize: $J = L(x, g(f(n(x))))$, where n adds noise. This forces the network to differentiate noise from signal.
    - Contractive autoencoders minimize: $J' = J + \sum \frac{\partial f}{\partial x}$. This forces the encoder to be smooth: similar inputs get similar outputs.
    - Predictive autoencoders: $J = L(x, g(h)) + L'(h, f(x))$. Instead of optimizing g and h simultaneously, optimize them alternately.
- Another solution is to train a discriminator network D which outputs a scalar representing the probability the input is generated.

Representation Learning

The idea is that instead of optimizing $u = f(x; \theta)$, we optimize:

$$u = r_o(f(r_i(x); \theta))$$

- r_i and r_o are the input and output representations, but the idea can apply to CNN, RNN and other models.
    - For example the encoder half of an autoencoder can be used to represent the input r_i.
- The hope is that there are other representations that make the task easier.

- These representations can be trained on large amounts of data and understand the base data.
- For example: words is a very sparse input vector (all zeros, with one active).
  - There are semantic representations of words that is easier to work with.

Practical Advice

- Have a good measure of your success.
- Build a working model as soon as possible.
- Instrument and iterate from data.

Appendix: Probability

- Probability is a useful tool because it allows us to model:
  - Randomness: truely random system (quantum etc).
  - Hidden variables: deterministic, but we can't see all the critical factors.
  - Incomplete models: especially relevant in chaotic systems that are sensitive to small perturbations.
- It is useful for reading papers and more advanced machine learning, but not as critical for playing around with a network.
- Probability: $P(x, y)$ means $P(\mathrm{x} = x, \mathrm{y} = y)$.
- Marginal probability: $P(x) = \sum_y P(\mathrm{x} = x, \mathrm{y} = y)$.
- Chain rule: $P(x, y) = P(x|y)P(y)$.
- If x and y are independent: $P(x, y) = P(x)P(y)$.
- Expectation: $\mathbb{E}_{x\sim P}[f(x)] = \sum_x P(x)f(x)$.
- Bayes rule: $P(x|y) = \frac{P(x)P(y|x)}{P(y)} = \frac{P(x)P(y|x)}{\sum_x P(x)P(y|x)}$.
- Self information: $I(x) = -\log P(x)$.
- Shannon entropy: $H(x) = \mathbb{E}_{x\sim P}[I(x)] = -\sum_x P(x) \log P(x)$.
- KL divergence: $D_{\mathrm{KL}}(P||Q) = \mathbb{E}_{x\sim P}\left[\log \frac{P(x)}{Q(x)}\right]$.
  - It is a measure of how similar distributions P and Q are (not true measure, not symmetric).
- Cross entropy: $H(P, Q) = H(P) + D_{\mathrm{KL}} = -\mathbb{E}_{x\sim P} \log Q(x)$.
- Maximum likelihood:
  - For p is data and q is model:
  - $\theta_{\mathrm{ML}} = \arg\max_\theta Q(X; \theta)$.
  - Assuming iid and using log: $\theta_{\mathrm{ML}} = \arg\max_\theta \sum_x \log Q(x; \theta)$.
  - Since each data point is equally likely: $\theta_{\mathrm{ML}} = \arg\min_\theta H(P, Q; \theta)$.
  - The only component of KL that varies is the entropy: $\theta_{\mathrm{ML}} = \arg\min_\theta D_{\mathrm{KL}}(P||Q; \theta)$.
- Maximum a posteriori:
  - $\theta_{\mathrm{MAP}} = \arg\max_\theta Q(\theta|X) = \arg\min_\theta -logQ(X|\theta) - \log Q(\theta)$.
  - This is like a regularizing term based on the prior of $Q(\theta)$.