Deep Learning and Stochastic Neural Models
Today’s lecture

• Shallow vs. Deep learning

• Stochastic neural models
  • Deep learning structures

• Varying network architectures
“Shallow” models

• Most models we used so far were “shallow”

\[ y = W \cdot x \]

• Single level of processing (PCA, linear classifier, ...)
  • Allows us to use simple representations of the input
    • e.g. linear features, likelihoods, etc.
Looking for depth

• The way we think is inherently hierarchical (i.e. “deep”)
  • e.g. remember the perceptual pathways?
    • Features of features of features, ...

• We don’t really make algorithms like that
  • Maybe we should!
Trying to get some depth

• How about we try multilayer PCA?
  \[ y = W_1 \cdot W_2 \cdot x \]
  • \( W_1 \) can contain eigenvectors of eigenvectors!
    • Is that a good idea?
  
• Not really, \( W_1 = I \) since \( W_2 \cdot x \) is whitened already
  • Also with other linear approaches, extra layers make no sense
    • They all collapse to a single linear transform
Revisiting the neural net

- An example of a “deep” architecture
Example classifier

- A “0” vs. “1” digit classifier
- 2-layer neural net with 2 hidden units and one output
Learned weights

- First layer is a “feature” transform
- Second layer is a simple classifier
The magic ingredient is the non-linearity

• At every layer we apply a sigmoid:

\[ y = g\left(W_1 \cdot g\left(W_2 \cdot x\right)\right) \]

• This allows us to meaningfully stack transforms
  • Hence to obtain a deep architecture
Multiple levels of features

- First layer contains low-level features, second layer contains mid-level features, ..., final layer is a classifier
Arbitrarily deep architectures

• We can stack as many transforms as we like
  • The goal is to find rich representations
  • Thus we make a "deep" model

• The goal is to get features of features of features of features of ...
  • "Like the brain does"
Whoa, wait a minute ... 

• You told us that 3 layers are enough for anything!
  • Why should we use more layers than that?
  • Aren't shallow models good enough?

• Yes, a shallow model is fine, but
  • There is no guarantee that you'll easily find the parameters!
  • Nor that you won't need a bazillion units
But there is a problem with depth

- Deep architectures have lots of parameters
  - In some cases in the billions!
- Typical neural net optimization becomes a problem

\[
\begin{align*}
x_i &= g\left( W_i \cdot x_{i-1} \right) \\
\delta_i &= \left( W_{i+1}^\top \cdot \delta_{i+1} \right) \cdot g'(W_i \cdot x_{i-1})
\end{align*}
\]
Problems with backpropagation

- Through many layers gradient becomes too small
  - We can’t propagate errors too far back

- Lots of local minima (lots of parameters!)
  - Even with shallow networks this can be a problem

- Biologically it’s a stretch
  - Does the visual cortex influence the retina?
  - Do we really have target values?
Desiderata

• We need a stable learning procedure

• We like biological plausibility
  • Computations should be mostly local
  • We like distributed systems

• We thus need to move away from backprop methods
A digression

• A bit on stochastic neural networks
  • A different structure from what we’ve seen so far
  • Hopfield and Boltzmann models

• Strong influences from physics and neuroscience
  • Recently resurgent models
The Hopfield/Boltzmann networks

- Auto-associative memory
  - Learns patterns by finding stable equilibria

- Fully connected & recurrent
  - All nodes have binary states \{0,1\}

- Model can learn to recall patterns
Energy minimization

- Parameters: State $s_i$, Weight $w_{ij}$, Threshold $\theta_i$
- After assigning patterns we minimize model “energy”

$$E = -\sum_{i<j} w_{i,j} s_i s_j + \sum_i \theta_i s_i$$

- Painful optimization problem!
  - Gradient solution to Hopfield
  - Stochastic solution to Boltzmann
The Restricted Boltzmann Machine (RBM)

• A more manageable form of Boltzmann machines
  • Two-layer, no intra-layer connections

• Visible and hidden nodes
  • Visible nodes represent known data
  • Hidden nodes are used for internal representation

• Easier to train and very useful for many tasks
Getting a handle on RBMs

• Define a probability of the network energy:

\[ E(v, h) = -a^\top \cdot v - b^\top \cdot h - h^\top \cdot W \cdot v \]

\[ P(v, h) = \frac{1}{Z} e^{-E(v, h)} \]

• \( Z \) is a partition function and \( a, b \) are node biases

• We also define the node probabilities

\[ P(h_i \mid v) = g\left(b_i + \sum_j w_{i,j} v_j \right), \quad P(v_i \mid h) = g\left(a_i + \sum_j w_{i,j} h_j \right) \]
Contrastive Divergence learning

- Maximize product of all $P(v)$
  - 1) For a training sample $v_1$ compute hidden node probabilities
  - 2) Generate a hidden layer activation vector $h_1$ from above
  - 3) Go back and generate a new input vector $v_2$ based on $h_1$
  - 4) Go forth and generate a new activation vector $h_2$ from $v_2$
  - 5) Update corresponding $w$ using: $\Delta w \propto v_1 h_1 - v_2 h_2$

"Positive gradient" \hspace{1cm} "Negative gradient"
So what does this do?

- Example on digit data
- 28×28 visible nodes
  - Set each pattern to a digit
- 100 hidden nodes
- \( W \) is 784×100
  - i.e. 100 “basis” functions
Back to deep learning

- The RBM is a shallow learner

- But we can use it to connect multiple layers
  - Treat each layer in a multi-layer input as an RBM

- The big idea: Train locally, group globally
  - This helps computational complexity and is biologically plausible
Deep Belief Networks (DBN)

- A stack of multiple RBMs
  - A deep generative model
- Initial weights are set using RBM training
- Further refinement using backpropagation
Greedy learning

- **Step 1:** Train an RBM for first layer
  - Visible nodes are the inputs

- **Step 2:** Fix hidden layer
  - Pretend it’s visible and train next layer

- **Step 3:** Keep going
So what is it good for?

- We can learn complex representations of data
  - Learn a deep model with multiple feature levels

- We can learn to classify
  - Use visible nodes to represent classes

- Example simulations on digit data:
  - Hinton's Neural Network Simulation (Generative)
    - Demo: [https://www.youtube.com/watch?v=KuPaiOgiHk#t=47s](https://www.youtube.com/watch?v=KuPaiOgiHk#t=47s)
    - 10 labels / 2000 l3 units / 500 l2 units / l1 500 units / 784 pixels
Some well-known press exposure

- A billion weights network trained on 10M YouTube frames
  - 1,000 machines for 3 days!
- Conclusion: YouTube has lots of cats! :)
  - and that we can get some great features that way

The cat neuron

The human body neuron
Helping out backpropagation

- We can also use this approach to learn large networks
  - e.g. a multiple layer classification network

- Use greedy learning to find initial values for weights
  - Treat each layer set as an RBM

- Once trained use as initial values for backprop
The importance of a good start

- Find “sensible” weight values
  - Don’t start from irrelevant points
- Starts from a space that is well-tuned to the data at hand
- Reduces the amount of required computations
Ugh, that RBM business is difficult ...

• There’s no reason to stick to RBMs
  • The math is a little tricky and the optimization costly

• Instead we could use another type of “shallow” learner
  • But it has to have a structure conducive to what we want
    • E.g. overcomplete ICA so that we can have more output nodes

• Or we can use an “autoencoder”
Autoencoders

- A very simple approach to designing a shallow non-linear feature extractor
- Try to learn an identity mapping
  - but we won’t make it that easy!
  - We won't give it enough resources
Ways to constraint an autoencoder

• Restrict the number of hidden nodes
  • Creates an information bottleneck
    • Resulting in an informative low-rank representation

• Go to higher dimensions but use sparsity
  • Creates informative “bases” and projects to high-D space

• Add some structure to the form of the layers
  • e.g. orthogonality, independence, etc.
Example case

• Running on digit data
Noisy autoencoders

- Find a “robust” representation
  - But stochastically corrupt the input
  - e.g. adding noise, removing random bits, transform it in non-linear ways, etc.

- Now the input is not always the same as the output
  - We need robust features that map all the noisy inputs to the proper output
Noisy autoencoders for enhancement

- We can also use noisy autoencoders to clean signals
  - Learn to predict a desirable output for a noisy input

- Can be used for multiple enhancement tasks
  - Removing noise, recovering higher resolution, ...

- Generate noisy/clean training data and learn a network
Toy example: Speech denoising

- Trained on 30sec inputs
  - Speech + street noise
  - Known speaker
  - Takes 30sec to train
    - on a laptop (2-3sec with GPU)

- Parameters
  - 1024pt spectra
  - 1 hidden layer, 100 nodes
  - Leaky ReLU activations
Runtime denoising

- Very lightweight process
  - ~300x real-time
    - 0.01sec in this case

- Works better than NMF
  - But can’t generalize to new noise types!

*Input mixture*  *Predicted output*
Stacked autoencoders

- Similar to DBNs
  - Multiple-layer architecture
  - Train each layer separately
    - Stack them all in the end

- Can also be used to classify
  - Once trained, add a classification layer and refine with backprop
So what’s new from the 90’s?

- Cynical view: Not much
  - We have more data and better processors
    - This allows us to train more useful models
  - The RBM/DBN business might have been a distraction …

- But, we now have many more tricks of the trade
  - Better activations, smarter training strategies, many little tricks to assist better convergence, more elaborate models than before, …
Dropout for better training

• Drop-out training
  • Randomly “turn off” units at every training iteration
    • Usually 50%
  • Puts pressure on units to be useful

• Results in a more robust networks
  • Equivalent to training multiple nets with shared weights, but slightly different connectivity patterns
Modern activation functions

- **Rectified Linear Units (ReLU’s)**
  - Instead of a sigmoid use: \( y_i = \max(0, x_i) \)
  - Much faster since there is minimal computation
    - Leaky versions: \( y_i = \max(\varepsilon, x_i) \)
    - Noisy versions: \( y_i = \max(0, x_i + n_i) \), ...

- **Softplus**: \( y_i = \log(1 + e^{x_i}) \)
  - “Softer” version of ReLU
Moving past gradient descent

• Stochastic Gradient (with optional momentum)
  • Use random batches, update using: $\Delta w_t = \nabla f(w_t) + \mu \Delta w_{t-1}$
  • Nesterov momentum: $\Delta w_t = \nabla f(w_t + \mu \Delta w_{t-1}) + \mu \Delta w_{t-1}$

• Rprop
  • Use gradient sign, make steps using adaptive learning rate/momentum
  • RMSprop: Normalize individual weight learning rates by the running average of past gradient magnitudes

• Adagrad / Adadelta
  • Learning rate-free approaches

• …
What difference does it make?

- Alec Radford’s excellent training animations:
Model Compression

• Is being shallow necessary?
  • Remember that three layers should be all we need

• Using a deep structure tends to facilitate training
  • Despite the introduction of many more parameters

• Train shallow networks to mimic trained deep networks
  • Train a deep network, generate a lot of outputs for random inputs and use them as training data for a shallow network
Getting more into signals

- As we’ve seen before we care about time!
  - That’s what makes a signal

- What we have so far is time-agnostic
  - Therefore a bad idea for signals

- How can we add some temporal structure?
Recursive Neural Networks

- Use node outputs as inputs
  - From anywhere to anywhere

- Some problems
  - Can form an unstable system
  - Can struggle with large data
    - More on RNNs on Friday

- RNNs are Turing complete!
  - In theory they can compute anything!
FIR Neural Networks

• Extend the scope of a neural net
  • Instead of weights, use FIR filters
    \[ x_i^l = \sum_j w_{i,j} \ast x_j^{l-1} \]

• Convolution is still linear
  • The usual backprop approach still works

• Good fit for temporal prediction tasks!
Convolutional Networks

- Generalizing the FIR filter idea
  - Unit weights are 2D filters (or 3D, 4D, ...)

- Max-pooling
  - For each neighborhood of filter outputs, keep only the max value
Convolutional networks for vision

- We extract deep features that make sense
  - These features are the filters at each layer
  - Also results in state of the art recognition!
And there’s much more ...

- **Good news:**
  - Very powerful and flexible approaches
  - Reasonable biological plausibility
    - And computability implications

- **Bad news:**
  - Too flexible, picking right structure is very much an art
  - Still very large/slow when it comes to training
Running deep learning today

- You need data, GPUs and software for it
  - GPUs and software are easy to get!

- Software takes care of learning/deriving models:
  - [http://deeplearning.net/software/theano/](http://deeplearning.net/software/theano/) ← Very general
    - [http://keras.io](http://keras.io) ← Bit more structured
  - [http://caffe.berkeleyvision.org](http://caffe.berkeleyvision.org) ← Vision oriented
Recap

- Deep learning concepts
- Stochastic shallow networks
  - Boltzmann machine
- Common deep architectures
  - DBNs, Autoencoders, convolutional networks
More material

- “Deep Learning”, the book:
  - http://goodfeli.github.io/dlbook/
  - (no PDF but you can read online)

- Learning Convolutional Feature Hierarchies for Visual Recognition

- A Fast Learning Algorithm for Deep Belief Nets
Next lecture

- More on deep learning

- Recurrent neural net models
  - RNNs, LSTMs, GRUs, etc

- Learning and classifying time-series