Machine Learning
Ali Bramson
9/28/2016

News of the day: head and neck cancers tricky because don't want to irradiate these areas. Have 700 images of patients with procedures, have which patients have aggressive side effects, build classifier for which areas safer to irradiate.

Linear regression equation- features and weights of inputs into neuron, and activation function (sigmoid) makes logistical regression. Magic of neural networks comes from stacking neurons together in layers. Get output of neurons… +1 always there to keep bias term going. Outputs go into next layer. Have many parameters to optimize… 3 inputs, 4 neurons gives 12 parameters to optimize.

How do we train neural networks?
First go all the way through, then go back and evaluate errors of middle layer nodes.

Gradient Descent: given a cost function (error- y is true layer, h() is what is coming out))- use this to modify weights in the whole network. Point of GD is to move weights in way to have highest impact. New weight is old weight + rate of change… need that equation for each connection between layers, inputs, etc. Same concept as in linear/logistic regression. Keep weights down so it doesn't overfit. Linear combination so calculate derivatives…

Rule of thumb- bad idea to initialize weights between 0 and 1. Takes a long time if use random initialization. For training, a good initialization range is: (equation on slide with +/- sqrt of 6 over nin + nout + 1)…so makes a very narrow range, narrower than 0 and 1.

Problems of NN: How many layers are enough to solve a problem and how many hidden units should we use per layer? Training complexity increases as we increase units (helped if avoid a full interconnection). Something called dropout – in a random step, take a random unit and eliminate it (connections, weights, everything) and do that N number of times and it works well to try to reduce the number of units in your network. Elephant in the room is called "Vanishing Gradient"- it's a problem.

If you work for a big company, youre like the CEO and your friend owns smaller store. Your friend has problem because his profit goes down and he only has 2 employees and he tells them its their fault, and each gets 50% blame. But now your big company (like Walmart), places blame on top 2 managers, and they pass it off to local managers under them, and they pass it down to all the workers….so many Walmart, so many cashiers, each worker shares only a very very small share of fault for profits going down….so they don't need to change their behavior that much, but guys at top are who is getting looked at. Vanishing Gradient is this problem.

Our small errors (delta) will become even smaller as we send the share of errors backward through the network to other layers. If we have many layers, we are going to end up with really

small gradients- errors get diluted; units don't need to really change anything…. Basically updating nothing. This will show as negligible updates in the gradient descent equations. This problem starts happening after 4 layers…neurons at beginning layers won't learn anything. In 2006, people came up with new paradigm to train network: pre-training on individual layers. The autoencoder.

The autoencoder is one of the many architectures of NNs. In the autoencoder we don't use the labels in the dataset…its an unsupervised learning algorithm so we don't run things like testing and training. Only has one hidden layer. Output and input are the exact same thing.

Say we have an 8x8 image-> make flat, 64 pixels…send each to autoencoder…interconnects fully…goal of autoencoder is to create exact same image on other side. Why train an algorithm to give something very similar? Key point is that the number of units in this autoencoder layer is less than number of inputs…its taking input and compresses it to less and then it has to decompress it again into the same original number. Lots of information that isn't needed…figure out what the main properties are that are needed…autoencoder learning important features of the image/data/input. Was a huge breakthrough.

MNIST Dataset: training set of 60,000 examples and a test set of 10,000 examples. Each digit 28x28 image (784 pixels). Each digit has a label that identifies which digit it represents (9 labels).

1998 to today list of papers with accuracy (# mistakes) on MNIST.... LeCun is MNIST guru and you can see during the years the accuracy going down. Started using Boosted trees. Then SVMs…get low results. Using NN there was a jump down… now people basically only using NN and recently people are using conv. NN. If you come out of new architecture for NN, for this dataset, need to get lower than this accuracy. For MNIST accuracy is a good measure…toy dataset, that's why people use it. Good to test on….balanced dataset, with about same number of each label… so yeah, if you can beat this number, you can write a paper (0.23) in machine learning.

Example on lecun.com - Can squeeze, rotate to 40 degrees, change weight of stroke and it still works well… with SVM the algorithm breaks the second it stops looking like a four or gets rotates, squeezed, etc. NN are translation invariant. Rotation invariance depends on what you're classifying. At some point the 4 won't look like a four anymore though, if rotate too much…but planetary sci could be good because a lot of features (like craters) are fairly symmetric.

Train an autoencoder with MNIST…examples for 10 hidden units and 80 hidden units. Try to compress all numbers info into 10 units, or compress to 80 and then expand it. Each perceptron has 784 weights because fully connected. Take weights with others, make matrix and reshape it back to your 28x28 image…reshape weights to see how weights look. Can only do for first layer. Treat weights as if a value of pixels, reshape to 28x28 and get what's shown on the slide…what each unit is learning. With combinations of these weights, could basically create numbers.

If we train an autoencoder and plug it in a NN then train things just work…why does it work so well? People not really sure.

Deep Nets: have input, place autoencoder step between input and classification. Train back forth back forth with gradient descent…wont get amazing results. Learn autoencoder with first layer of units, train it to get 4 neurons and the weights they learn from the autoencoder and then plug them into your network to initialize the weights. Then initialize the rest of the layers as you did before. Do a couple passes and then new thing you did with plugging autoencoder results in brings accuracy from 80% to 95%....just by changing initialization terms of first layer. Will allow these to change for training but wont change a ton because of the vanishing gradient thing. But just initializing them this way in deep nets (nets with more than 4 layers) gives much better results.

Representation learning conference – every paper tries to explain why this approach works so well. So important there's its own conference right now.

Could also use autoencoder to initialize weights of each layer by running first layer after autoencoder step and use output of the first layer's weights to input into next autoencoder, and go down the line to use to train each layer of autoencoder with previous autoencoder/layer. Cant train inner layer with autoencoder without doing layers before it.

When is sample too small for deep nets? In Leon's experience, less than 500 samples. Related to samples and number of classes though so MNIST works well with only 3 layers because only 9 labels.

Bring laptops next class.