Neural Network

Jiatong

Handling Non-Linear Data

- Option1: add feature by hand that make the data separable
  - Requires feature engineering
- Option 2: Learn a small number of additional features that will suffice
  - Today
- Option 3: Kernel Trick

Motivation

- Where do features come from
  - We build them by hand
- What if we wanted to learn features
  - Goad: learn features that given linearly separable data
- After learning features apply usual linear classifier

Perceptron: Graphical Representation

Why Non-Linear

- Generalized linear classifiers
  - Start with linear function w*x
  - Pass the output through a non-linear function y=h(w*x)
  - What is h
    - Nonlinear function (logistic tanh sign relu)

Multi-layer perceptron

- Perceptron: a training method for generalized linear classifiers
  - Training method for linear classifiers
  - Minimize the error of the training data
  - Chain Multiple perceptron together
  - Update rule
    - $w^{j+1} = w^j + \nabla f(x, y)$

Example

Output
Y
...
Y_K
K(D+1) weights
Hidden Layer
Z_1
Z_2
...
Z_D
D(M+1) weights
Input
X_1
X_2
X_3
...
X_M
18

Network Terminology

- Input nodes: x
- Output node: y
- Hidden nodes: z
  - This network has 1 hidden layer
  - 2-layer network (2 layers to learn)
- H for hidden nodes are called activation function
- H for output depends on task
  - Identity for regression
  - Logistic for classification

Classification Objective

- Define an error function and minimize
- Cross-entropy error function
  - $E(w) = -\sum(y_i \ln(\hat{y}_i) + (1 - y_i)\ln(1 - \hat{y}_i))$
- This arises naturally when we consider a logistic probability model and take the negative log likelihood

Regression Objective

- For regression we use the sum of square error
  - $E(W) = \frac{1}{2}\sum(\hat{y}_i - y_i)^2$
- If we assume a Gaussian model for y, the error function arises from maximizing the likelihood function
  - We saw the same thing for linear regression

Training

- Prediction is relatively easy
- Learning is where the magic
- Strategy: compute the gradient of the objective function
  - Similar to perceptron
  - Gradient based update

Go deeper

Output
Y
Hidden Layer
Z_1
Z_2
...
Z_F
Hidden Layer
Z_1
Z_2
...
Z_E
Hidden Layer
Z_1
Z_2
...
Z_D
Input
X_1
X_2
X_3
...
X_M

Motivation: why go deeper

- Why is it easier to train
  - Deep architectures can be representationally efficient
- Deep representations allow for a hierarchy, non-local generalizations
  - Possible with shallow networks, but fewer computational units for the same function in deep network
- Deep Nets: Multiple levels of latent variables allow combinatorial sharing of statistical strength

NN Packages

- Pytorch
- Tensorflow
- Caffe
- Mxnet
- CNTK
- Chainer

Modularity

- The core building blocks can be combined to create new models
  - Library provides core building blocks
  - User combines them into models
- Don’t we need to write out the gradients for our models

Softmax $$\hat{y}_k = \frac{\exp(b_k)}{\sum_{l=0}^{K} \exp(b_l)}$$

Loss (cross entropy)
$$J = \sum_{k=1}^{K} y_k \log(\hat{y}_k)$$

Output (softmax)
$$\hat{y}_k = \frac{\exp(b_k)}{\sum_{l=1}^{K} \exp(b_l)}$$

Output (linear)
$$b_k = \sum_{j=0}^{D} \beta_{kj} z_j$$

Hidden (sigmoid)
$$z_j = \frac{1}{1 + \exp(-a_j)}$$

Hidden (linear)
$$a_j = \sum_{i=1}^{M} \alpha_{ji} x_i$$

Input
$$x_i, \forall i$$

Compute Gradients

- We need a way to compute gradients of arbitrary network structure
- Make the gradient computation efficient

Automatic Differentiation

- Write the objective function as a combination of smaller building blocks
  - Algorithm will automatically compute the derivative for use in learning

- Wikipedia description: a set of techniques to numerically evaluate the derivative of a function… [which] exploits the fact that every computer program… executes a sequence of elementary arithmetic operations (addition, subtraction, multiplication, division, etc.) and elementary functions (exp, log, sin, cos, etc.). By applying the chain rule repeatedly to these operations, derivatives of arbitrary order can be computed automatically

Backpropogation

- Backprop of errors in MLPs is a special case of automatic differentiation
- AD allows us to compute gradient of arbitrary functions as long as we can specify how the function breaks down into parts
  - Computation graph: a DAG where each node is a variable in the function
  - Where would we get such a graph?

Forward Propogation

- Forward computation
  - Write out the network structure as a directed acyclic graph of computations
  - “Computation graph”
- Visit each node in topological order
  - For each variable ui with inputs v1 … vn
  - Compute ui=g(v1,…vn)
  - Store the result at the node
- You now have network output, as well as all internal nodes

Training: Chain Rule

- Given y=g(u), u=h(x)
- Chain rule:
  - $\frac{dy_i}{dx_k} = \sum_{j=1}^{J} \frac{dy_i}{du_j} \frac{du_j}{dx_k} \forall i, k$

Backward Propagation

- Initialize all partial derivatives
  - $\frac{\partial y}{\partial u_j} = 0, \frac{\partial y}{\partial y}$=1
- Visit each node in reverse topological order
- For variable $u_i = g(v_1, \ldots, v_N)$
  - We know $\frac{\partial y}{\partial u_i}$
  - Increment $\frac{\partial y}{\partial u_i} \frac{\partial u_i}{\partial v_j}$
  - We will choose g to make this easy
- Return $\frac{\partial y}{\partial u_i} \forall u_i$

Training

Backpropagation

Simple Example: The goal is to compute $J = \cos(\sin(x^2) + 3x^2)$ on the forward pass and the derivative $\frac{dJ}{dx}$ on the backward pass.

Forward

$J = cos(u)$

$u = u_1 + u_2$

$u_1 = sin(t)$

$u_2 = 3t$

$t = x^2$

Training

Backpropagation

Simple Example: The goal is to compute $J = \cos(\sin(x^2) + 3x^2)$ on the forward pass and the derivative $\frac{dJ}{dx}$ on the backward pass.

Forward	Backward
$J = cos(u)$	$\frac{dJ}{du} \mathrel{+}= -sin(u)$
$u = u_1 + u_2$	$\frac{dJ}{du_1} \mathrel{+}= \frac{dJ}{du}\frac{du}{du_1}, \quad \frac{du}{du_1} = 1 \qquad \frac{dJ}{du_2} \mathrel{+}= \frac{dJ}{du}\frac{du}{du_2}, \quad \frac{du}{du_2} = 1$
$u_1 = sin(t)$	$\frac{dJ}{dt} \mathrel{+}= \frac{dJ}{du_1}\frac{du_1}{dt}, \quad \frac{du_1}{dt} = \cos(t)$
$u_2 = 3t$	$\frac{dJ}{dt} \mathrel{+}= \frac{dJ}{du_2}\frac{du_2}{dt}, \quad \frac{du_2}{dt} = 3$
$t = x^2$	$\frac{dJ}{dx} \mathrel{+}= \frac{dJ}{dt}\frac{dt}{dx}, \quad \frac{dt}{dx} = 2x$

Climbing

−1.6
−1.8

Why SGD Work

- Even non-convex SGD will converge
  - Converge == arbitrarily small gradients
- The best solution isn't the only good solution
  - Big data: with enough data, you can find reasonable solutions
- Learning rates
  - We have lots of adaptive algorithms
- Random restarts/extensive hyper-parameter optimization
- Regularization

Data

- We have lots and lots of data
- Labeled data
  - Still not much labeled data for many tasks
- How can we make use of unlabeled data for neural network training

Unsupervised Training Insights

The hidden layers of the network are learning representations of the data

- Only the last layer of the network directly depends on knowing the label y
- Can we learn better representations on unlabeled data, and then prediction parameters on labeled data?

Problem: Vanishing gradients

- The gradient for an edge at the base of the network depends on the gradients of many edges above it
- The chain rule multiples many of these gradients together

Auto-Encoders

Activation Function

Sigmoid / Logistic Function

$$\text{logistic}(u) \equiv \frac{1}{1+e^{-u}}$$

- A new change: modifying the nonlinearity
  - The logistic is not widely used in modern ANNs

Alternate 1: tanh

Like logistic function but shifted to range [-1, +1]

Activation Function (Cont'd)

- A new change: modifying the nonlinearity
  - reLU often used in vision tasks

Alternate 2: rectified linear unit

Linear with a cutoff at zero

(Implementation: clip the gradient when you pass zero)

Activation Function (Cont'd)

- A new change: modifying the nonlinearity
  - reLU often used in vision tasks

Alternate 2: rectified linear unit

Soft version: log(exp(x)+1)

Doesn't saturate (at one end)
Sparsifies outputs
Helps with vanishing gradient

Activation Function (Cont'd)

Overfitting in Deep Networks

- Very large networks can easily overfit
- Large numbers of parameters
- More and more with each layer of the network
- What strategies can we use to prevent over fitting?

Regularization

L1 or L2 regularization on network parameters

- Straightforward based on previous uses of regularization

Different parameters call for different regularization

- We may have more complex network architectures
- We could apply different types of regularization to different types of parameters

Early Stopping

- Traditional training: run until convergence
- Early stopping
  - Measure performance on held out evaluation set
  - Stop when held out accuracy no longer improves
  - This is “early”, i.e. before convergence
  - Measuring “no improvement” can be tricky

Dropout

- Instead of modifying objective function (regularization), modify the network structure

Dropout

Why does it work

- Noise robustness
  - Dropout is similar to making random perturbations in the input
  - These small changes shouldn’t change the prediction for an input
  - Think “reduces variance”

Why does it work

Prevents neuron co-adaptation

- Neurons, especially near the top of the network, suffer from co-adaptation
- The neurons take on values that rely on other neurons for correct predictions
- Multiple points of failure: any single neuron failing can invalidate the others
- Dropout prevents co-adaptation by randomly removing some of these neurons
- Network is forced to learn more robust features that are useful with many different random subsets of other neurons

Why does it work

Ensemble training

- Recall from boosting: we can combine many classifiers to produce better output
- During training, each random permutation is one of 2n possible neural networks
- All networks share weights so number of parameters is still O(n2)
- Dropout training is like training a collection of 2n ”thinned” networks with extensive weight sharing
- Test time: approximate the average prediction of all networks by using a single network without dropout

Dropout: Advantages

- Works really well for a wide variety of network types and domains
- Very easy to implement with no modifications in training
  - Some minor details (omitted) are important
- Drawbacks
  - Need to select dropout rate
  - This depends on the types of parameters and network structure

Still Unanswered Questions

- Lots of motivations help us to understand why they work
- But they are mostly ad hoc explanations for an empirically useful training procedure

More data

- Pre-training on large amounts of data
- Randomly permute input data to create a larger data set
- Add additional objective functions to increase supervision

Reduce connection to local regions

Example: 1000x1000 image
1M hidden units
Filter size: 10x10
10M parameters

Reuse the same kernel everywhere

Because interesting features (edges) can happen at anywhere in the image.

Share the same parameters across different locations:

Convolutions with learned kernels

Learn multiple filters.

E.g.: 1000x1000 image
100 Filters
Filter size: 10x10
10K parameters

Detail

If the input has 3 channels (R,G,B), 3 separate k by k filter is applied to each channel.

Output of convolving 1 feature is called a *feature map*.

This is just sliding window, ex. the output of one part filter of DPM is a feature map

Summary of a typical convolutional layer

Doing all of this consists one layer.

- Pooling and normalization is optional.

Stack them up and train just like multi-layer neural nets.

Final layer is usually fully connected neural net with output size == number of classes

Recurrent Neural Networks

Feed-forward NN

$$\mathbf{h} = g(\mathbf{V}\mathbf{x} + \mathbf{c})$$

$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{h} + \mathbf{b}$$

Recurrent NN

$$\mathbf{h}_t = g(\mathbf{V}\mathbf{x}_t + \mathbf{U}\mathbf{h}_{t-1} + \mathbf{c})$$

$$\hat{\mathbf{y}}_t = \mathbf{W}\mathbf{h}_t + \mathbf{b}$$

Recurrent Neural Networks

$$\mathbf{h}_t = g(\mathbf{V}\mathbf{x}_t + \mathbf{U}\mathbf{h}_{t-1} + \mathbf{c})$$
$$\hat{\mathbf{y}}_t = \mathbf{W}\mathbf{h}_t + \mathbf{b}$$

How do we train the RNN's parameters?

Recurrent Neural Networks

$$\mathbf{h}_t = g(\mathbf{V}\mathbf{x}_t + \mathbf{U}\mathbf{h}_{t-1} + \mathbf{c})$$

$$\hat{\mathbf{y}}_t = \mathbf{W}\mathbf{h}_t + \mathbf{b}$$

Recurrent Neural Networks

- The unrolled graph is a well-formed (DAG) computation graph—we can run backprop
  - Parameters are tied across time, derivatives are aggregated across all time steps
  - This is historically called “backpropagation through time” (BPTT)

“Read and summarize”

$$\mathbf{h}_t = g(\mathbf{V}\mathbf{x}_t + \mathbf{U}\mathbf{h}_{t-1} + \mathbf{c})$$
$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{h}_{|\boldsymbol{x}|} + \mathbf{b}$$

Summarize a sequence into a single vector.
(This will be useful later…)

sj1

Example: Language Model

$$p(tom \mid \langle \mathtt{s} \rangle) \times p(likes \mid \langle \mathtt{s} \rangle, tom) \times p(beer \mid \langle \mathtt{s} \rangle, tom, likes) \times p(\langle /\mathtt{s} \rangle \mid \langle \mathtt{s} \rangle, tom, likes, beer)$$

Slide 57

sj1 shi jiatong, 7/24/2020

Language Model Training

Training Challenges

$$\mathbf{h}_t = g(\mathbf{V}\mathbf{x}_t + \mathbf{U}\mathbf{h}_{t-1} + \mathbf{c})$$
$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{h}_{|\boldsymbol{x}|} + \mathbf{b}$$

What happens to gradients as you go back in time?

$$\frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_2}\frac{\partial \mathbf{h}_4}{\partial \mathbf{h}_3}\frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{h}_4}\frac{\partial \mathcal{F}}{\partial \hat{\mathbf{y}}}\frac{\partial \mathcal{F}}{\partial \mathcal{F}}$$

Vanishing Gradients

- This means that long-range dependencies are difficult to learn (although in theory they are learnable)
- Solutions
  - Better optimizers (second order methods, approximate second order methods)
  - Normalization to keep the gradient norms stable across time
  - Clever initialization so that you at least start with good spectra (e.g., start with random orthonormal matrices)
  - **Alternative parameterizations: LSTMs and GRUs**

Alternative RNNs

- Long short-term memories (LSTMs; Hochreiter and Schmidthuber, 1997)
- Gated recurrent units (GRUs; Cho et al., 2014)
- Intuition instead of **multiplying** across time (which leads to exponential growth), we want the error to be **constant**.

h t-1
h t
h t+1
A
A
tanh
σ
σ
tanh
σ
X t-1
X t
X t+1

$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

“Deep” LSTMs

- This term has been defined several times, but the following is the most standard convention

Sequence to Sequence transduction with RNNs

Recurrent Neural Networks (RNNs)

$$\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t)$$

$$\mathbf{c} = \mathrm{RNN}(\boldsymbol{x})$$

What is a vector representation of a sequence $\boldsymbol{x}$?

RNN Encoder-Decoders

$$\mathbf{c} = \mathrm{RNN}(\boldsymbol{x})$$

What is the probability of a sequence $\boldsymbol{y} \mid \boldsymbol{x}$?

Cho et al. (2014); Sutskever et al. (2014)

RNN Encoder-Decoders

$$\mathbf{c} = \mathrm{RNN}(\boldsymbol{x})$$
$$\boldsymbol{y} \mid \mathbf{c} \sim \mathrm{RNNLM}(\mathbf{c})$$

What is the probability of a sequence $\boldsymbol{y} \mid \boldsymbol{x}$?

Cho et al. (2014); Sutskever et al. (2014)

I'm
hungry
</s>
ich
habe
Hunger
</s>
<s>
I'm
hungry

Encode anything as a vector!

Q&A