dl-notes

Ian Gomez

December 29, 2021

Contents

1 CNN

1.1 Convolution Operation

- Comes from signal processing the ideas is that we can use an average of multiple measurements to get a more accurate representation of the signal.
- However instead of the continuous version we tend to use the discrete version which is Sum(x(a)*w(t-a)) (continuous is just integral)
- This can be viewed as a dot product using specific matrices like the toeplitz matrix

1.2 Motivation

- CNN benefits from sparse interactions this reduces runtime and allows for similar power due to sparsity since we only apply weight to certain patches to get a higher level representation instead of having to apply it for every feature
- Parameter sharing reduces the amount of parameters that need to be learned since they are shared along the input
- Equivariance changes in the input are reflected as changes in the input e.g. if patches move around in the picture then the convolution applied to that patch is translated in the same way
- Convs are not equivariant to all changes but are equivariant to translation

1.3 Pooling

- Reduces the amount of paramters by summarizing local areas
- This allows the operation to be invariant to small changes in the output since even small pixel shifts will get absorbed by the pooling.

1.4 Theoretical Analysis/Pros and Cons

- Theoretically we can view a CNN as a model which assumes an infinitely strong prior that we should be invariant to small changes. and learn from local areas
- This is because we can view the matrix of parameters we learned as sparse due to the convolution operation.
- This can cause underfitting like any prior since if we make assumptions that don't represent the data our model will be unable to learn.
- Some cases of this are when we need to preserve precise spatial information where pooling the features can lead us to underfitting.
- When trying to learn from distant feature it may not be useful to use convolutions which impose the prior that we learn from local patches
- Finally we can only compare CNN's to CNN's due to the effects of permutation

1.5 Variants of Convolution

- When working with images we usually use multi-channel convolution where the linear operations are not guaranteed to be commutative
- They are only commutative if each operation has the same # of input and output channels
- We can add stride to act as a downsample (reduces how fine grain we produces features)
- We can also add padding to prevent the shrinking of the image
- Full padding allows for us to revisit the border as frequently as other pixels

- We can also used an unshared convolution or locally connected layers which has different parameters but chooses which locality to use aka convolution with no param sharing
- Can also used tiled convolutions
- Gradient Derivation On page 357

1.6 Structured Outputs

- We do not just need to output class labels or regression values one thing we can do is create structured outputs that for example predict probability masks for each pixel in an image

1.7 Data Types

- Can process inputs with varying spatial extents. As opposed to fixed window of FNN's
- Can work for 1-D 2-D 3-D both single and multi channel

1.8 Random or Unsupervised Features

- Using non supervised features in convolutions helps to reduce training time
- One way to obtain these kernels is random init/or desining by hand. Finally one can use unsupervised approaches to learn these features
- For unsupervised approach one way is to use k-means on small patches and using the centroid as convolution kernels
- This is similar to pretraining the conv layers and then relearning the last layer which leads it to be similar to a convex optimization problem in the case of a logistic regression or SVM
- Random filters also work well
- One way for choosing architectures quickly is to first evaluate the performance of convolutional networks with random kernels and then train the final layers and evaluate performance
- We can also do greedy layer-wise pretraining (train first layer in isolation then second layer etc.)

- An example of this is the convolutional deep belief network
- While unsupervised methods may be good most stuff is fully supervised however unsupervised can provide good results due to regularization or reduced computational costs of the leraning rule.

1.9 Neuroscience basis for conv nets

- Studies showed that early visual systems responded to certain light patterns while other light patterns did not make these systems respond

2 RNN

- Utilize parameter sharing to allow the model to learn different parts of the sequenece independent of location.
- Similar to the idea introduced by TD networks however those have shallow connections due to localized neighbors
- Can be applied to multiple dimensions and even have connections to previous time steps (Granted that all time steps are observed prior) Allowing the computation graph to contain cycles

2.1 Unfolding Computational Graphs

- Unfolding the computational graphs using recursive definitions we arrive at an acyclic representation of the computation graph.
- We can furthermore paramterize this function using an input x where we see we include the input and a summary of the hidden state to create the output sequence.
- Finally we use extra features to extract the information out of the final h and make a prediction
- This hidden state is a lossy representation of the task-relevant aspects of the sequence till time T
- It is guarenteed to be lossy since the sequence is indefinitely long while the hidden state is fixed in size
- Due to selectively choosing which aspects of information to store. The hardest task for RNN's is when we need h to recover the input sequence like an autoencoder

- By having the recurrent unfolding structure we are allowed to generalize to unseen sequence lengths due to parameter sharing. While also needing fewer samples.
- The unfolded graph helps to be explicit by showing how information flows forward and how it flows backward when computing gradients

2.2 Recurrent Neural Networks

- RNN's have many different considerations on architectural design many-to-many, many-to-one, recurrence based on output
- RNN's using many-to-many design can simulate any turing machine
- Due to the nature of recurrence backprop is expensive requiring O(T) runtime for forward pass which cannot be reduced by parallelization since the input needs to be processed one after the other.
- It also has a memory cost of O(T) due to needing the states for backwards computation
- This computation is called backprop through time

2.3 Teacher-Forcing and BPTT

- The output recurrent architecture is strictly weaker since the output needs to capture the past while also matching the output target
- However with this recurrent NN we can benefit from the fact that we can parallelize our loss function since we can use the ideal output as the input of our hidden state
- We can use teacher forcing which is the idea of using the ground truch as input at time t+1
- Models can utilize both BPTT and teacher forcing
- if used in open-loop mode the network outputs which are then used as inputs may not be like the test distribution
- Some ways to mitigate this is to use teacher-forcing only after predicting a certain length of sequence, or to randomly choose generated values as input

2.4 Computing the gradient

- Gradient derivation
- Essentially compute gradient onto the hidden layer where u need to sum the gradient of the output of that layer and the gradient of the next hidden state in relation to the current hidden state after that compute the gradient in respect to the params of the hidden state

2.5 Recurrent Networks as Directed Graphical Models

- We tend to model probability distributions with these models (e.g. cross-entropy for multiple classes or unit gaussian for regression)
- One thing that is important is that when modeling this using MLE that P(y|x...) becomes P(y|x...,$y_{1...yt}$-1) when we feed in the outputs of previous time steps as inputs.
- this is important because many models tend to exclude any connections that deal with y (e.g. assuming markov property). However, the RNN inherently captures this through the previous bullet
- This also shows that the RNN is efficient in describing the probability distribution over the dataset. One naive implementation for describing this probability distribution is using a table of probabilities where each entry is the probability of the class as time T this is $O(k^T)$ where the RNN is O(1) since it doesnt scale with time T and is a constant set by the user
- However while it is parameter efficient. It is not easy to optimize this model
- There are also some assumptions made in this model. One of which is that the hidden state is not dependent on t when moving between time steps. One way to alleviate this is to add t as an input however then the model would need to learn this correlation
- Finally the last thing to cover is how to sample from distribution. We can just sample normally from the conditional however we need to know when to stop the sequence. There are a few ways to sample one way is to create a symbol which means stop. Another is to add a bernoulli output that predicts whether or not to stop (this is a sigmoid output trained with BCE). Finally we can predict an integer that outputs the sequence length and then we sample n many times

2.6 Modeling Sequences Conditioned on Context with RNN

- In the last section we view the DAG as only representing a joint distribution, however the original formulation allows us to represet our distribution in a more powerful way. A joint distribution conditioned on the X variables (inputs).
- As previously discussed any P(y; W) can be reinterpreted as P(y|W). we can furthermore extend it as P(y|X) where w is a function of x
- We can provide the extra input either each timestep, as the initial hidden state or both.
- In the case where we have a fixed input parameter we can concatenate our input value with our parameters in a way that models P(y|X)
- We can also receive x as a sequence where in this case we assume $P(y_{1...yT}|x1..x_T)$ is conditionally independent and gets decomposed to $Prod_T(P(y_t|x1...x_T))$
- To get rid of this independence we input the output into the next timestep
- This however has one restriction in that x and y need to be the same sequence length (This is where we introduce seq2seq)

2.7 Bidirectional RNNs

- Previously we assume causality in that the value of y is computed strictly from previous x (and y)
- However there are case in which we need to know the whole sequence's context
- This can be very important for speech recognition tasks where the correct interpretation of the current phoneme depends on the next few phonemes.
- In the case of two words that have ambiguity (e.g. buy and by) we may even need to look at previous word level outputs to remove ambiguity
- this is also true of handwriting recognition and many other seq2seq learning tasks

- These are extremely useful for handwriting recognition, speech recognition (may be useful for my intern project), and bioinformatics
- The output of the bidirectional RNN allows us to create a model which is sensitive to context around t but also utilizes context from farther away and variable sequence length
- This can be further extended to images which utilize an RNN that goes in 4 directions (Up D L R)
- RNN's composed in this way allow for long range dependencies. Across the image

2.8 Encoder-Decoder (seq2seq)

- Comes up in many applications speech recogniition, machine translation, question answering
- We often call the input the context and we want to produce a summary of that context for the decoder RNN
- In seq2seq the two RNNs are trained to maximize $P(y_{1...yny} | x_{1..xny})$
- Encoder produces context C which is usually a function applied to the final H_{nx} of the encoder
- Decoder is conditioned on that fix length vector to generate the output sequence. This allows nx != ny. In this architecture both RNNs are trained jointly to maximize the above function
- As previously shown RNNs can receive context either as the hidden state or as input (There is no constraint that $H_{enc.shape} == H_{dec.shape}$)
- The size of the encoder RNN is very important since it must be large enough to encapsulate all of the input sequence
- Thus they decided to make C a variable length sequence rather than a fixed size vector and used Attention to learn how to associate elements of sequence C to the output (Explained in 12.4.5.1)

2.9 Deep recurrent networks

- The idea is self explanatory, however by empirical results it seems beneficial since we are only learning a shallow representation
- We can think of lower layers as transforming the input to an easier to generalize representation.
- We can also go a step further and add an MLP for every deep block (input to RNN layer, deep RNN block, deep output block) however this hurts optimization (obviously)
- If we use a MLP for state to state transitions then the distance between hidden variables becomes 2x however, we can use skip connections to mitigate this
- Deep RNNs can be structured either hierarchically, With deep MLPs in input to hidden, hidden to hidden, and hdden to output. And to mitigate difficulty of optimization we can add skip connections to mitigate the problem of optimization

2.10 Recursive Neural Networks

- a recursive nerual network generalize the recurrent structure into a recursive tree.
- Good for processing data structures NLP and computer vision
- The depth of a RNN can be reduced from T to O(log(T)). where depth is the number of compositions of nonlinear operations (e.g. in RNN amount of hidden transitions is linear as opposed to logrithmic in in recursive)
- One option is to use a balanced binary tree. Another can be lead by domain expertise like for language using parse trees to lead the tree struture
- Ideally it would be good for the learner to learn the tree structure
- Computation at each node does not need to be the traditional neruon computation (e.g. linear combination + non-linear activaton). can use tensor + bilinear forms

2.11 The Challenge of Long-Term Dependencies

- As show previously gradients can either shrink or explode due to multiple multiplications of the jacobian in backpropagation
- To exemplify this if we view each recurrent step as h_t = W.T h_t-1 the equation become $h_t = W^{t.T} h_0$
- if we assume W.T can be decomposed into a Q L Q.T (eigendecomposition of a symmetric matrix) then the formula become h_t = Q.T L^t Q h_0
- Therefore the eigenvalues of the matrix decomposition determine whether or not the gradients explode or vanish as sequences get longer. However, all eigenvalues of 1 will not be affected
- However since it is very unlikely for all values to be == 1, it is almost certain that this will explode/vanish.
- Also besides that we in order for RNN's to store memories robust to perturbations the RNN must enter the space where gradients vanish
- This doesn't mean that long term memories are impossible to form, moreso that it will take a very long time to learn as opposed to short term depndencies, and are sensative to fluctuations in short term dependencies.
- Empirically it was shown that at sequence lengths of 10 or 20 the gradient started to approach 0
- While there have been some remedies that have been proposed in the upcoming sections. It is probably one of the main challenges in DL.

2.12 Echo state networks

- The most difficult parameters to learn are the mappings from h_t-1 and x to h_t
- One way that was proposed is to set recurrent weights manually/programtically such that it does a good job of capturing past weights
- These are called reservoir computing, in that it draws from a resvoir of previous outputs

- Its similar to a kernel in that it maps an arbitrary length sequence to a fixed length vector.
- One benefit of this view is that it is essentially a linear regression which is convex when using certain losses
- The spectral radius (largest absolute valued eigenvalue) will maximize the seperation of perturbations thus being a good candidate for seperating differences
- Something is contractive when a linear mapping always shrinks h. When the spectral radius is < 1 it is always contractive (explaining the thing b4 about having RNN's needing to explore vanishing gradient space to be resistant to small perturbations)
- Even if the values were complex values what matters most is the complex absolute value in which case magnitudes >1 explode and <1 shrink
- The idea for echo networks is to fix weights that have high spectral radius but doesn't explode due to the effect of saturation of things like tanh

2.13 Leaky Units and other strategies for multiple time scales

- One strategy is to incorporate leaky units and other things which allows us to view things with multiple time scales (e.g. short term and long term).

2.14 Skip Connections through Time

- Same idea as skip connections in CNNs by adding recurrent connections that operate every d steps it decreases decay to only t/d as opposed to t. gradients can still explode in time t

2.15 Leaky units and a spectrum of different time scales

- Another way to get derivatives close to one is to have linear self-connections and a weight near one on these connections.
- An example of this is having a running average of mu_t and some value v_t and using the function u_t = a*u_t-1 + (1-a)*v_t. in this case a is a linear self-connection since a either fully remembers the previous value and when a is near zero it updates the information.

- When hidden units have learn self-connections they behave similarly to running averages like mentioned above and are called leaky units.
- There are two strategies. Sampling them during initialization time or learning them.
- Having these at different time scales allows us to better model long term dependencies

2.16 Removing Connections

- Anotherway is organizing the rnn at multiple time scales.
- The idea is to remove length-one connections and replace them with longer connections.
- This differs from skip connections since those add edges.
- You can also mix in leaky units to this.

2.17 The LSTM and other Gated RNNs

- Using the idea of leaky units we want to have the option to choose when to keep and when to remove information from the hidden state.
- Ideally we don't want to do this ourselves and instead we make the NN learn when to dynamically forget or remember information this is what gated units do.

2.18 RNNs (Not that many notes because I have studied this a lot)

- Aside from the default equations for the LSTM update rules you can also include the hidden state into the context for computing the gates.

2.19 Other gated RNNs

- GRU (prolly not that many notes)
- uses a single gating unit using the update gate and reset gate which chooses to ignore certain aspects of the input and update determines the full updating rule

- Some other things can be designed with this as a base (sharing forget gates across multiple hidden units). Global gate could also be used
- No variant between LSTM and GRU are proven to provide clear superior model.

2.20 Optimization for long term dependencies

- After some research it has been shown that second order derivatives may dissapear at the same time as first order derivatives. These second order optimization methods are also difficult to optimize and are attracted to saddle points. It has been shown that first order algorithms tend to work better in practice.
- It goes to show that sometimes simpler easy to optimize algorithms are better than more powerful algorithms

2.21 Clipping Gradients

- Due to the problem of cliffs in the loss curvature it may cause gradient updates to massively overshoot and lead them to worse optima.
- Therefore we use gradient clipping to prevent this one way is based off of clipping it elementwise and another is to normalize it with the norm of the gradient
- e.g if $||g|| > v$ $g = g*v/||g||$
- While theoretically the norm based way may seem better due to perserving direction both work about as well empirically.
- Elementwise clipping can be good in the case of Inf/Nan where moving in a random direction will help to get out of the bad area.
- Also elementwise clipping introduces a heuristic bias to the estimator of the overall gradient that is attempted to be performed in minibatch gradient descent.

2.22 Regularizing the gradient

- While clipping helps explosion it doesnt help vanishin
- As shown previously we want the gradient values to be close to 1 so that we can minimize the possibility of gradient vanishing

- Therefore we can regularize the gradient such that it can be about as large as the gradient that flowed into the current step of the computation graph.
- When combined with gradient clipping it is beneficial in helping RNN learn long term dependencies.

2.23 Explicit memory

- NN's are good at storing implicit memories (subconscious). However struggle with explicit memories and require seeing something many times to remember something.
- Some people have designed systems to act as working memory (explicit memory which stores information relevant to a goal)
- Hinton believes that it is more important to see how NN's change inputs in regard to the time steps is more important that the actual mapping in terms of determining if reasoning actually exists.
- Some models like memory networks and Neural Turing Machines try to adress this.
- NTM's use read write actions from memory cells via attention mechanisms.
- Since its difficult to find exactly which address to access it does it parallel on multiple addresses using a softmax to figure out which cells to access.
- Instead of storing scalars we store vectors in explicit memory since its more costly, and also acts like content based addressing e.g. when we remember stuff its like we specifically access the part of our memory that related to certain content.
- Attention is important.