CS7140: Advanced Machine Learning **Spring 2020**

Lecture 8: Structured Prediction

Lecturer: Rose Yu *Scribe: Luke Roberto*

8.1 Structured Prediction

Structured prediction is a class of algorithms that focus on models that have a structured output space. If we have a function $f : \mathbf{X} \to \mathbf{Y}$, then a structured domain Y is one that has regularities or admits certain properties/invariants. The motivation arises from the types of problems we have been solving (for supervised learning) of the form:

$$y^* = f(x) = \text{argmax}_y g(x, y)$$

If we choose this function to optimize over to be $p(y|x)$ when our model is probabilistic, then this corresponds exactly to MAP estimation. These problems in general are quite difficult to solve, and for a finite set of $\mathbf{X}$ and $\mathbf{Y}$ is NP Hard (there is a combinatorial number of solutions to search over). We know that the observed data, $\mathbf{Y}$, is often quite structured. In order to make learning more efficient, we can encode inductive priors into our model structure so that the algorithm does not need to learn these computations purely from data.

8.1.1 Applications of Structured Prediction

Figure 8.1 shows a few examples of structured prediction in practice.

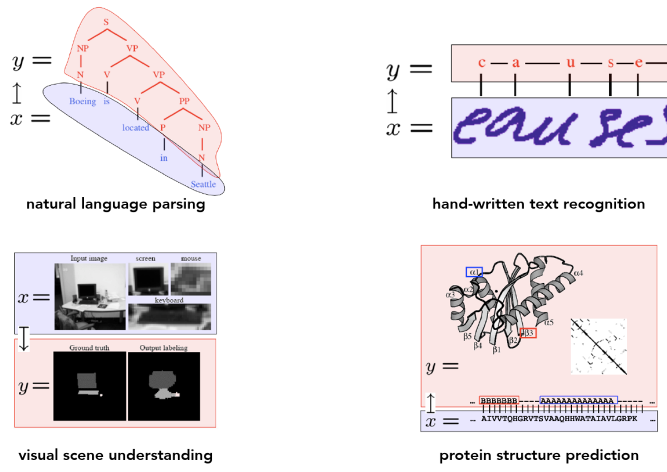

Figure 8.1: Top Left: Taking a sentence and predicting the semantic structure (NLP). Top Right: Inferring letters from hand written text. Bottom left: Performing image segmentation based off the content of an image/image crop. Bottom right: Predicting folded protein structure from the fundamental components, like amino acids and hydrogen bonds.

8.2 Classical Structured Prediction

The first class of algorithms we will survey are *Classical Structured Prediction.*

8.2.1 Graph Cuts

Figure 8.2 shows a typical problem in computer vision called Image Segmentation. The goal is to try and segment out particular objects or entities that are part of the *Foreground* (the lambs), while the rest is the *Background* (the grass). This can be formulated as a binary graph cuts problem. The problem setup is we

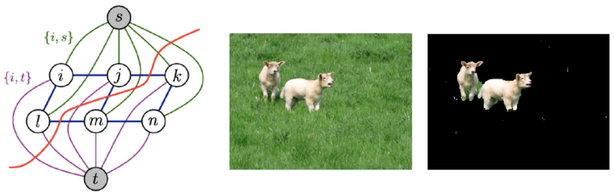

Figure 8.2: Caption

are trying to find an assignment of each node such that there are either part of the source (upper cut in left figure) or the sink (lower cut). The algorithm tries to find a global maximizer for the following quantity:

$$g(x, y; w) = \sum_{i \in V} \log p(y_i|x_i) + w \sum_{(i,j) \in E} C(x_i, x_j)[y_i \neq y_j]$$

The first term in the function to be maximized is the sum over the color model for a particular vertex. We want to maximize the likelihood of a label given the color of a pixel. The second term adds weight to pairs of nodes in a different class based on their intensity, which maximizing will make sure they are as different as possible.

These types of problems suffer from the curse of dimensionality due to large feature spaces, and inability to tackle problems in a probabilistic way. Optimality is not necessarily a feature that we want in prediction algorithms, as our training data usually has a good amount of noise in it.

8.2.2 Local Search

Local Search methods attempt to solve the curse of dimensionality in structured prediction problems by searching over small regions of the solution space in an iterative fashion. In doing this, they do give up their ability to find the optimal solution for a given problem.

Figure 8.3 shows an abstract version of a local search-based algorithm. $N(y^0)$ represents the neighborhood that will be searched starting from initial seed y^0. The next step searches in the neighborhood of the next solution y^1. Solutions are chained together iteratively until there are no neighbors that yield a better solution.

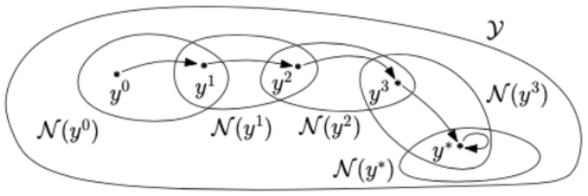

Figure 8.3

8.2.3 Branch and Bound

Another class of algorithms are called *Branch and Bound*, which are a special case of divide-and-conquer style algorithms. These algorithms trade-off worst-case complexity in order to efficiently find solutions for practical, typical problem instances.

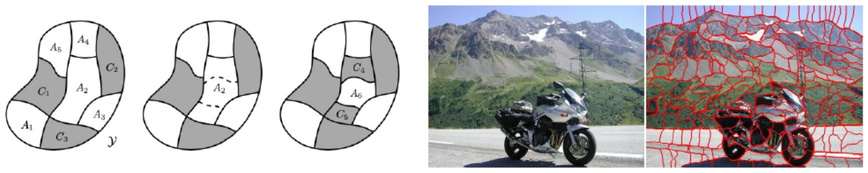

Figure 8.4: Right: The image is divided into multiple sets. Branch and Bound iterates over these sets and uses them to create representative superpixels.

The image on the right of figure 8.4 shows the branch and bound strategy for an abstract problem. We partition the space of solutions into sets of active and closed nodes. For each set, we try to compute a bound for the set as a whole, so we do not have to enumerate every solution. The algorithm is an abstract scheme, and so it requires the user to define the problem-specific selection, branching, and bounding functions.

8.2.4 Learning to Plan

Search-based planning algorithms can be useful for problems where we have a cost function that encodes things we care about in a solution. Figure 8.5 shows an example of vehicle routing in multiple training images (3 to the left). We can use the dataset to learn the cost function directly from the examples of paths that we care about. Once we have this function, we can simply run an off-the-shelf search algorithm like A^* and augment it with our learned loss function.

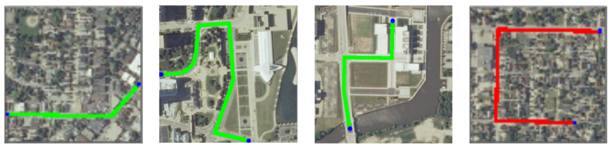

Figure 8.5

8.2.5 Lagrangian relaxation

In gradient-based optimization, we often run into problems that have hard (equality) constraints. In order to solve problems of this nature, lagrangian relaxation uses the method of lagrange multipliers in order to turn a constrained optimization problem into an unconstrained optimization. Assume we have some objective function, $J(x)$, subject to constraint $C(x) = 0$. The lagrangian relaxation is given by:

$$\hat{J}(x) = J(x) + \lambda C(x)$$

An example of this related to content in the course would be performing MAP inference on a markov random field (MAP-MRF). The energy function for the system potential can be used for the objective function, and then constraints can be added to ensure its a proper probability distribution and all the marginals are consistent:

$$E(y;\theta,\mu) = \sum_{m,n} \theta_{mn}\mu(y_m, y_n) + \sum_n \theta_n \mu(y_n)$$
$$\text{s.t} \quad \sum_n \mu_n(y_n) = 1, \sum_n \mu_{m,n}(y_m, y_n) = \mu_m(y_m)$$

Which can be converted into a loss for an unconstrained optimization problem:

$$J(y) = \sum_{m,n} \theta_{mn}\mu(y_m, y_n) + \sum_n \theta_n \mu(y_n) + \lambda_1(\sum_n \mu_n(y_n) - 1) + \lambda_2(\sum_n \mu_{m,n}(y_m, y_n) - \mu_m(y_m))$$

8.3 Deep Structured Prediction

In the previous examples, we saw many classical algorithms perform tradeoffs on optimiality, constraint satisfaction, worst-case complexity, and robustness. Modern methods in deep learning propose learning purely through data to update model information. The main issue with using a standard densely connected architectures is that it assumes no structure in the output space of our data. The number of possible outcomes is so large it can be quite difficult for a network to learn a proper representation to effectively do prediction. The next few sections will detail inductive biases that can be encoded in neural network architectures in order to improve training.

8.3.1 CNNs

Convolutional Neural Networks are a specific architecture that arose from encoding special structure in images. Images are data structures that are 2D or 3D (multiple channels), where each pixel represents a particular intensity. CNNs are not restricted to only 2D and 3D data (see applications section).

8.3.1.1 Structure

There are two main structural elements present in images that we want to encode in our neural network: *Locality* and *Translation invariance.* Locality refers to the statistics in a neighborhood of pixels, they are quite similar. A network should take this into account by only pooling local information in order to distill features from an input. Translation invariance refers to the global statistics of the image. Larger scale features, like eyes on a face or petals on a flower, should be activated in a similar way no matter where they are located in the image. The relative locations of these features are what matter, not their absolute locations. Figure 8.6 shows a representation of these inductive priors we want to encode in our network.

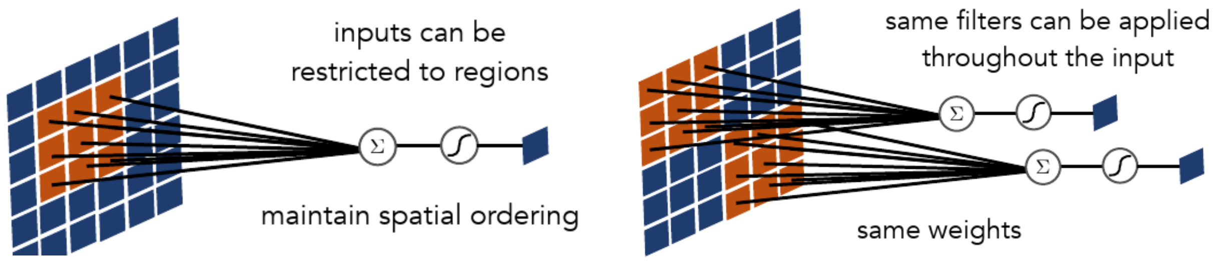
inputs can be restricted to regions
maintain spatial ordering
same filters can be applied throughout the input
same weights

Figure 8.6: **Locality**: Nearby pixels tend to contain stronger patterns, be similar, and vary in related ways. **Translation Invariance**: Image patches tend to share characteristics and are combined in particular ways, so the relative position of features matters the most.

The type of operator that satisfies these types of conditions we want to have on our processing is known as the *Convolution Operator*.

8.3.1.2 Convolution

The convolution is a common operation in the signal processing and dynamical systems analysis. We can think of it as a "sliding sum" that takes a function and slides it across another, adding up all the overlap between them. Another way to think of it is that the functions are vectors, and we want to take the dot product of these vectors over over each lag. Here is the mathematical representation of the convolution for the 1D and 2D cases:

$$(f * g)(t) = \sum_i f(i) \cdot g(t-i) \quad \text{(1D Discrete)}$$

$$(f * g)(t) = \sum_i \sum_j f(i,j) \cdot g(x-i, y-j) \quad \text{(2D Discrete)}$$

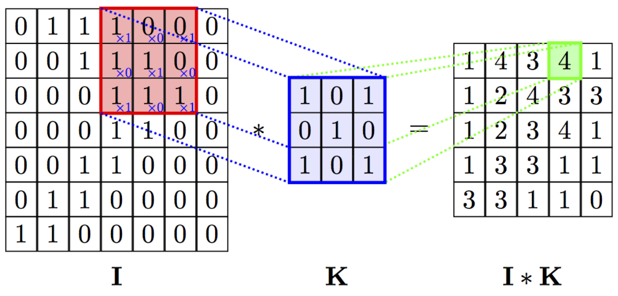

Figure 8.7

Often, the function that we are sliding is called the *Kernel* or the *Filter*. The resulting output is called a *Feature Map* or *Activation Map*. Figure 8.7 shows an example of convolution in practice when we are dealing with images. $\mathbf{I}$ is the input image and $\mathbf{K}$ is the filter to convolve. We take the dot product at each index of the input image in order to generate $\mathbf{I} * \mathbf{K}$.

8.3.1.3 Applications

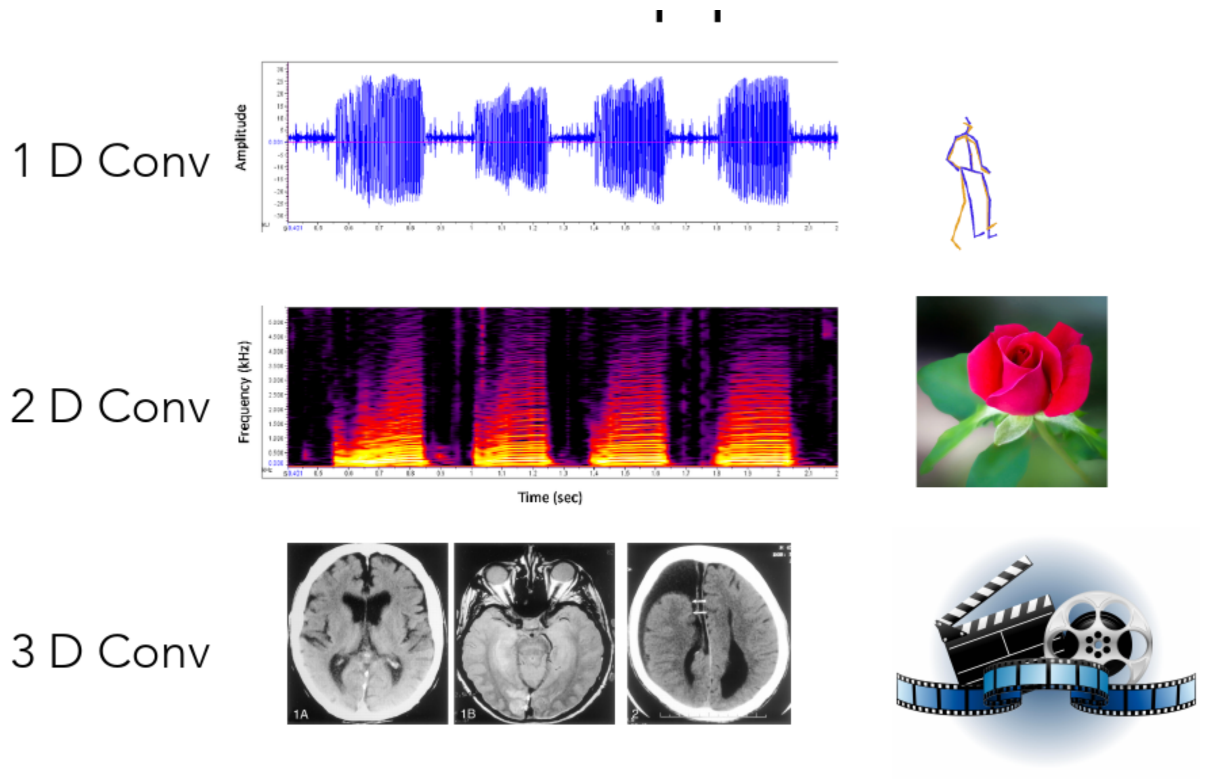

Figure 8.8: 1D Convolution: Audio data can be processed using 1D CNNs, for things like speech recognition or filtering. 2D Convolution: The left shows an image of a spectrogram. This data structure is a 2D array of time and frequency bins that can be used to analyze fourier transforms of signals in time. CNNs can distill useful features from these "images" in order to perform prediction. 3D Convolution: The left image shows slices of a brain scan, or volumetric data. A 3D CNN can perform convolutions in 3D space to filter for features. The right image alludes to the use of video data. That is 2 spatial dimensions and 1 time dimension that a 3D convolution can operate over.

8.3.2 RNNs

Similar to how we wanted to encode useful structure about image/spatial data into a neural network, another type of data is time-series data. The defining features of time series data is that data points very close in time should be similar in value (locality), that the absolute time in which something occurs is irrelevant (translation invariance), and that data should be processed sequentially (temporal ordering).There are many types of RNNs that solve a variety of issues that arise in training these types of networks, figure 8.9 shows some examples of cell topologies.

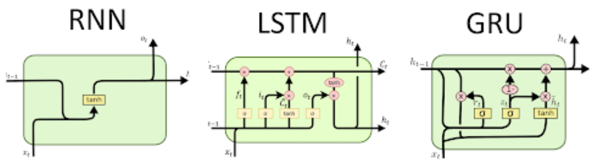

Figure 8.9

8.3.2.1 Structure

The main structural element of an RNN can be seen in figure 8.10. On the left we see that the input is passed into a hidden cell h that keeps an internal state of what it has seen up to the current point in time. We can unroll this to see what the structure looks like through time. The hidden state starts off at some initial value, and then gets updated once an input is passed through. The hidden state is then stored to be part of the next prediction in the sequnce, iteratively for all time (until it is reset).

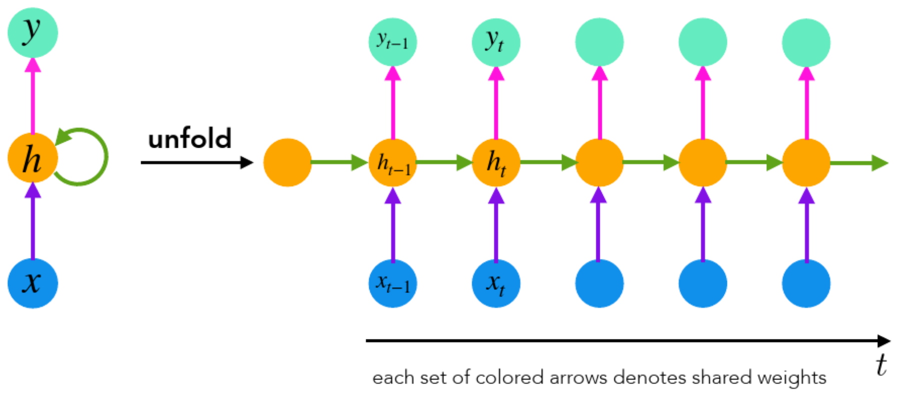

Figure 8.10

We can see that this encodes locality via the hidden state, as temporally close inputs will be processed with similar hidden states. The translation invariance is encoded with the fact that every input is process with the same network weights, only the hidden state will change during processing. The sequential processing is encoded by the unfolding of the RNN to take inputs through time.

One interesting parallel to note is to that of a Hidden Markov Model (HMM). The HMM graphical model is almost identical to an RNN without inputs. The key difference is that the RNN does not model distributions. All of its outputs are deterministic, so the same kinds of richness in modeling we get with probabilistic models can not be gained with these neural network architectures.

8.3.3 Backpropagation Through Time (BPTT)

One of the major issues with RNNs is how to properly train them. BPTT is essentially extending backpropagation to RNN training via the unrolling we discussed earlier. Figure 8.11 shows an how the gradients flow the simplified network from before.

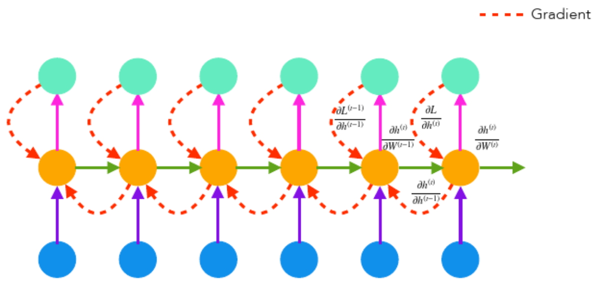
Gradient
$\frac{\partial L^{(t-1)}}{\partial h^{(t-1)}}$
$\frac{\partial h^{(t)}}{\partial W^{(t-1)}}$
$\frac{\partial L}{\partial h^{(t)}}$
$\frac{\partial h^{(t)}}{\partial W^{(t)}}$
$\frac{\partial h^{(t)}}{\partial h^{(t-1)}}$

Figure 8.11

If you start at the right of the diagram (last time step show), the first gradient to be accumulated is the loss with respect to the last hidden unit $\frac{\partial L}{\partial h^{(t)}}$. $W^{(t)}$ is introduced as copies of the network weights over time, but only get updated at step t. The gradient flows backwards in time, accumulating the new loss gradients from previous steps until the initial state can be updated.

This poses a difficulty in propagating gradients over a really long time horizon. The gradients being chained together in such a deep network with typically get quick small and provide no useful information to inputs far away in time (the vanishing gradient problem). There is also a large computational/memory footprint associated with storing large amounts of state for long time horizons. The most naive way to solve this issue would be to add skip connections across steps so that this accumulation does not pose as much of an issue. This increases amounts of computations and also requires more hyperparameters on windows sizes.

The best approach right now is to actually reform the cell itself into a trainable memory module. Examples of this can be seen in figure 8.9. LSTMs and GRUs essentially put gates inside of the cell so that the network is able to read and write from its hidden state based off of context. This allows it to keep information from longer time horizons.

8.3.3.1 Applications

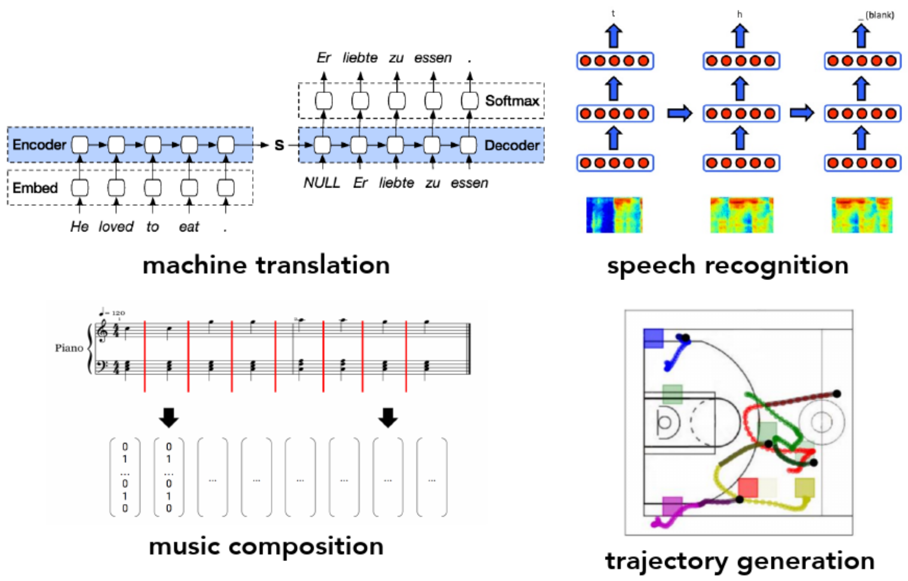

Figure 8.12: Top Left: Automatic translation from one langauge to another, feeding the sentence in sequentially and then having the RNN decode the hidden state into the translated sentence. Top Right: Taking in a sequence of spectrograms for different time intervals in order to recognize parts of speech. Bottom Left: Music composition, take in some image or sequence of features in order to predict the composition of a piece of music. Bottom Right: From an initial trajectory, try to predict what the trajectory will be a few steps into the future.

References

[1] Sebastian Nowozin and Christoph H. Lampert (2011), ”Structured Learning and Prediction in Computer Vision”, Foundations and Trends® in Computer Graphics and Vision: Vol. 6: No. 3–4, pp 185-365. http://dx.doi.org/10.1561/0600000033