Data C182 Designing, Visualizing & Understanding DNN
Fall 2024 Eric Kim, Naveen Ashish

Discussion 05

This discussion covers computer vision architectures and basic recurrent models.

1. Review of Vision Problems

For most of the class thus far, when we discuss applying neural networks in practice to vision applications, we have largely assumed an image classification task. That is, given an image, we let the network output the probabilities of the true label belonging to a variety of classes.

However, there are more types of standard computer vision problems. At the high level, we can roughly categorize the computer vision problems into three main categories (the 3R's of vision): **recognition**, **reconstruction** and **re-organization**. Recognition is about attaching semantic category labels to objects and scenes as well as to events and activities. Reorganization is about the partitioning of the image based on semantic information. Reconstruction is about obtaining the 3D information of of the scene that generated the images. Under these broad categories, we can further classify the problems into specific tasks.

Image Classification Given an image, we would like the network output the probabilities of the true label belonging to a variety of classes. This type of problem was the main focus of the course so far.

Object Localization Determine a *bounding box* for the object in the image that determines the class. In this type of problem, only one object is involved, and indeed, we know ahead of time that there is only one object class of interest in the image. Often, the bounding box objective may be simultaneously trained with the classification objective, resulting in a loss objective that is the sum of the two loss terms, the L_2 and the cross-entropy loss, respectively.

Object Detection Determine multiple objects in an image and their bounding boxes, with performance measured by *mean average precision* (mAP). There may be many objects, and several instances of the same object class (for e.g., several dogs) in the same picture. This means that, in contrast to image classification where the network only has to identify one object, the network has to predict a varying number of bounding boxes. In literature, object detection can be solved using R-CNNs (R-CNN, fast R-CNN, faster R-CNN, mask R-CNN).

Semantic Segmentation Label every pixel in the image. Here, we can naively run a CNN classifier for each pixel. However, better solutions, like `UNet`, exists in literature. *Semantic* segmentation means we do not worry about distinguishing between different instances of a class, in contrast to the aptly-named *instance* segmentation problem.

2. Image Classification: ResNet

Let's start with one of the most famous successes of deep learning: ResNet. A standard image classification convolutional neural net involves a bunch of convolutional layers, then a layer that converts from activations of shape (batch size, channels, height, width) to (batch size, some fixed dimension), followed by one or more affine layers. The final output of the affine layer is usually a vector of dimensionality equal to your number of classes that you're trying to classify the image into: e.g., if you're trying to classify hand-written digits, you would need 10 categories. You can then train this model with the cross-entropy loss discussed in lecture.

There are a few approaches for that intermediate conversion step: for example, you could use a flattening layer (which converts the intermediate activations from shape $(\text{batch size}, \text{channels}, \text{height}, \text{width})$ to $(\text{batch size}, \text{channels} \times \text{height} \times \text{width})$. Alternatively, you can pool over each filter map (one per channel) to produce an output of shape $(\text{batch size}, \text{channels})$. Think about why you might want one of these approaches over the other.

Regardless of the specific design choices, stacking up too many layers in your neural network can cause empirically bad performance. This is kind of a problem though: we need sufficiently deep neural networks to ensure they have enough expressivity, especially for more complex tasks that require the neural network to learn a very complicated mapping (e.g., from images to 1000+ categories).

Problem: Issues with Very Deep Neural Networks

What problems would a very deep neural network encounter? For simplicity, you can consider a very deep fully-connected network (i.e., just interwoven affine layers and ReLUs).

This is exactly the problem that ResNet set out to resolve. To do this, they introduce the concept of a residual connection (also called a skip connection or residual layer). For a given intermediate layer (or sequence of layers) $F(\cdot)$ and input to that layer x, rather than outputting $F(x)$, the residual layer outputs:

$$F(x) + x$$

Where the above expression uses element-wise addition. $F(\cdot)$ can be whatever kind of standard neural network layer you want it to be. The only restriction is that the input x and output $F(x)$ must be the same shape.

Problem: Residual Shapes

Why do the shapes need to be the same? If $F(\cdot)$ is one or more convolutional layers, how can we enforce that this is the case?

By adding residual connections to all their layers, the authors of the ResNet paper were able to train models significantly larger and deeper than all previous ones, achieving then state-of-the-art performance while suffering no gradient issues. Such connections are still regularly used today, including in modern Transformer-based vision or language models.

Problem: Why are Residual Layers Good?

What are some reasons why residual layers might aid in training very deep neural networks?

3. Recurrent Neural Network

The world is full of sequential information, from video to language modeling to time series data. In particular, we would like to model these sequences using neural networks, and solve some major types of tasks that we would like to solve with sequence models.

0.1 Types of Problems

- **One-to-one** problems take a single input x and produce a single output y. Problems like classification (takes an image as input, and produces a class label as output) and semantic segmentation (image as input, segmentation mask as output) fall under this category.
- **One-to-many** problems take a single input, and produce a sequence of output. Problems like image captioning (takes a single image as input, and produces a caption (a sequence of words) as output) fall under this category.
- **Many-to-many** problems take sequences of inputs and produce sequences of outputs. As shown in fig. 1, there are two main subcategories of many-to-many problems. In the first case, an entire input sequence is processed before producing an output sequence. An example of this is language translation (sequence of words in one language to sequence of words in another). On the other hand, some problems produce an output for each step in the sequence. An example would be controlling a robot (wherein the neural network takes in a sequence of observations and, for each one, must produce some robot action).

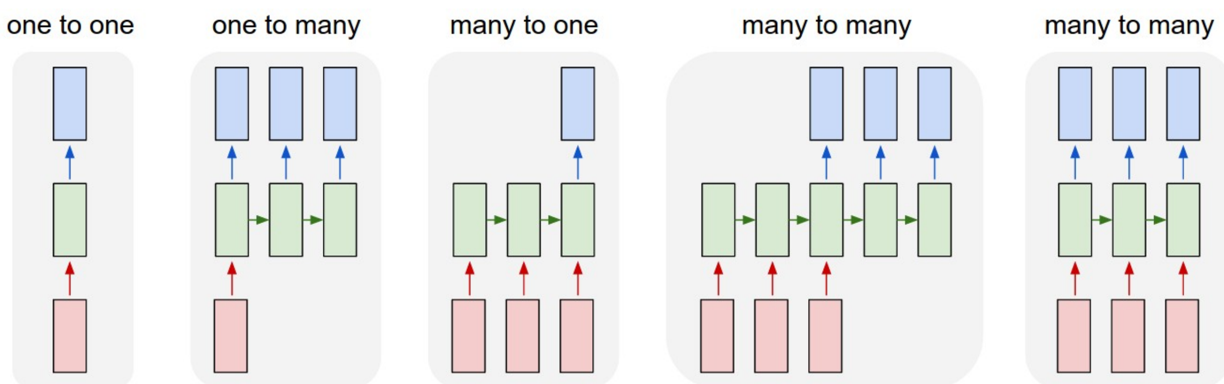

Figure 1: Types of problems we would like to solve using sequential models

0.2 Why the Recurrence?

As you read through this discussion worksheet, you don't process each word entirely on its own, but instead use your understanding from the previous words as well. Traditional neural networks do not have the capability to use its reasoning about previous events to infer later ones. For example, if we would like to classify what is happening at every frame in a movie, this can be framed as an image classification task where the network is provided the current image. However, it is unclear how a traditional neural network should incorporate knowledge from the previous frames in the film to inform later ones.

Recurrent neural networks (RNNs) address this issue, by using the idea of "recurrent connections." RNNs are networks with loops in them that allow information from previous inputs to persist as the network processes the future inputs. These recurrent connections allow information to propagate from "the past" (earlier in the sequence) to the future (later in the sequence).

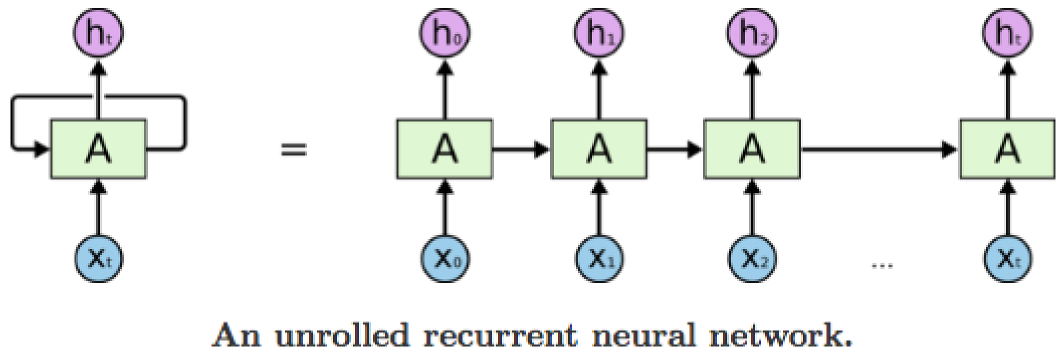

Figure 2: An example of a generic recurrent neural network. This shows how to "unroll" a network through time - instead of thinking about sequence modeling as a single network with shared weights

In Figure 2, we illustrate the RNN computation as it is unrolled through time. Each $i \in \{0, \ldots, t\}$ represents a new timestep in the network. By feeding in a state computed from earlier timesteps as an input together with the current input, information can persist throughout the time as the network "remembers" the past inputs it processed.

0.3 Vanilla RNN

In the following section, we will use the following notation. Denote the input sequence as $x_t \in \mathbb{R}^k$ for $t \in \{1, \ldots, T\}$, and output of the network be $y_t \in \mathbb{R}^m$ for $t \in \{1, \ldots, T\}$. In the following example, we construct a "vanilla" many-to-many RNN, consisting of a node that updates the hidden state h_t and produces an output y_t at each timestep with the following equations:

$$h_t = \mathsf{tanh}(W_{h,h}h_{t-1} + W_{x,h}x_t + B_h)$$
$$y_t = W_{h,y}h_t + B_y$$

where h_t is the time step of a hidden state (one can think of h_{t-1} as the previous hidden state), $W_{\cdot,\cdot}$ be the set of weights (for example, $W_{x,h}$ represents weight matrix that accepts an input vector and produce a new hidden state), y_t be the output at timestep t and B_h and B_y be the bias terms.

As for the shapes, if the hidden states h_t are of size d_h and inputs x_t are of size d_x, then $W_{h,h}$ and $W_{x,h}$ are (d_h, d_h) and (d_h, d_x) respectively, while B_t is of size d_h. For the output y, if it has shape d_y, then $W_{h,y}$ (which maps the hidden state to an output) must be of size (d_y, d_h) and the bias is of size d_y.

We can also represent it as the diagram below,

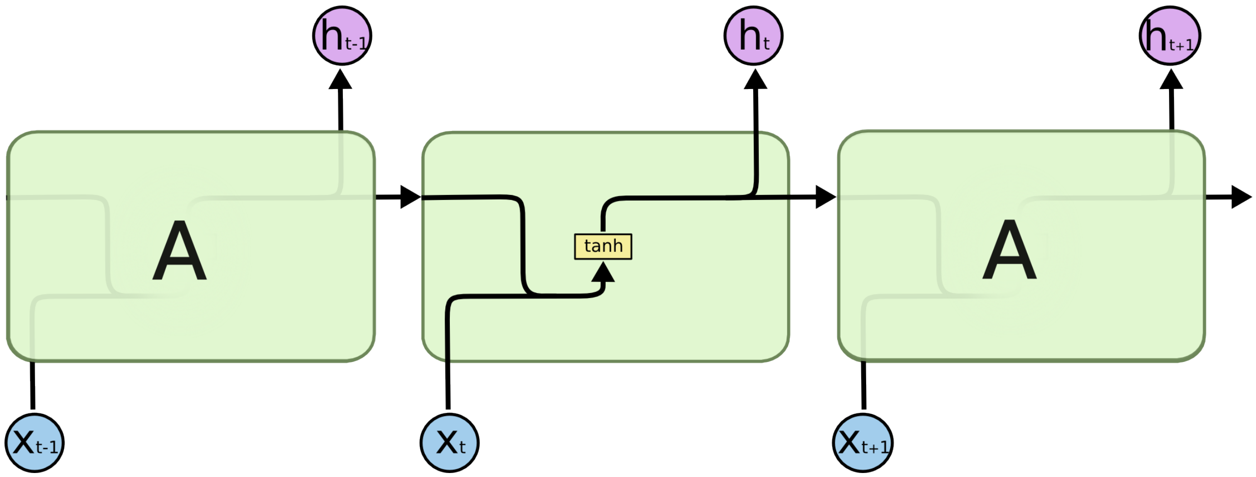

Figure 3: A simple RNN cell. As we can see by the arrows, we only pass a single hidden state from time $t-1$ to time t

In this vanilla RNN, we update to a hidden state "h_t" based on the previous hidden state h_{t-1} and input at the current time x_t, and produce an output which that is a simple affine function of the hidden state. To compute the forward (and backward) passes of the network, we have to "unroll" the network, as shown in Figure 2. This "unrolling" process creates something that resembles a very deep feed forward network (with depth corresponding to the length of the input sequence), with shared affine parameters at each layer. Our gradient is computed by summing the losses from each time-step of the output.

Problem: Gradients in Vanilla RNN

Why are vanishing or exploding gradients an issue for RNNs?

However, just as residual connections can be used to deal with vanishing gradients for feed-forward deep neural networks, it can also be used for recurrent networks! Many more sophisticated architectures, such as long short-term memory (LSTMs) and gated recurrent units (GRUs) effectively have a separate hidden state term that gets passed forward in time (roughly) linearly, just like how residual layers allow x to "bypass" a layer linearly. Those architectures introduce some other complicated operations, but the core idea is the same – as are the benefits, as such models can typically handle much longer sequential tasks.

Problem: Coding RNNs Up!

Complete the class definition, started for you below. Assume that your hidden state is a vector of size 4, your input x is a vector of size 5, and your desired output y is a vector of size 3.

```
import numpy as np

class VanillaRNN:
    def __init__(self):
        self.hidden_state = np.zeros((?, ?))
        self.W_hh = np.random.randn(?, ?)
        self.W_xh = np.random.randn(?, ?)
        self.W_hy = np.random.randn(?, ?)
        self.Bh = np.random.randn(?)
        self.By = np.random.randn(?)

    def forward(self, x):
        # Processes the input at a single timestep and
        # updates the hidden state
        self.hidden_state = np.tanh(...)
        self.output = np.dot(...) + ...
        return self.output
```

Problem: Forms of Sequential Inputs

You'll notice that we assume the inputs to the RNN, x_t, are vectors (since you're multiplying matrices with them). However, a lot of the time, your sequential data isn't easily represented by vectors. How can an RNN take the following as input: (1) a sequence of images (i.e., a video) or (2) discrete data (like sequences of letters or words)?

