Lecture Notes, Week 12: Neural Networks

1 Neural Networks

Neural Networks are an important class of models in machine learning and computer vision. Neural networks have produced, and continue to produce, impressive results in the broad spectrum of vision problems (*e.g.*, classification, object detection, segmentation, and many many others). As with any supervised learning, neural networks allow one to define a certain parametric model and then learn (optimize) parameters of that model based on data in the form of input-output pairs (*e.g.*, images and labels). In order to do this one typically needs to define a problem dependent objective or a *loss function* which measures the severity of miss-prediction for each example in the training dataset. In what follows, we will discuss the basic building blocks of the neural networks, their construction mechanisms and methodology for learning.

1.1 A neuron

The core building block of a neural network is a *neuron*. A neuron takes some number of inputs and produces a single output. A behavior of a neuron can be characterized by the following simple equation:

$$neuron(\mathbf{x}) = \alpha(\mathbf{w} \cdot \mathbf{x} + b)$$

where $\mathbf{x} \in \mathbb{R}^d$ are d inputs, $\mathbf{w} \in \mathbb{R}^d$ is a *weight vector* and $b \in \mathbb{R}$ is a scalar *bias term*. In other words, a neuron simply takes a weighted combination of the inputs and subtracts a constant, b, from this sum. The result of this operation, which is called pre-activation, is then passed into an *activation* function $\alpha(\cdot)$.

A number of choices exist for activation functions. For example, ReLU, which stands for Rectified Linear Unit has the following simple form:

$$\alpha(x) = \text{ReLU}(x) = \begin{cases} x <= 0 & 0 \\ x > 0 & x. \end{cases}$$

In other words, it just zeros out any negative value passed into it and leaves positive values intact. Another common and effective alternative is a Sigmoid function,

$$\alpha(x) = \text{Sigmoid}(x) = \frac{1}{1 + e^{-x}}.$$

Many other options for activation functions exist. An important property is that activation function has to be non-linear in order to ensure that the neural network that result from using neurons with such activations is an *universal approximator* – can approximate any function arbitrarily closely. For this to be the case, some additional conditions have to hold regarding the activation function and also the construction of the network. I talked about this in the lecture. Note, that the reason activation function cannot be linear is that it is easy to show that under this condition the neural network can only model linear functions irrespective of network topology, which is too restrictive.

The ReLU and Sigmoid activation functions have a number of important properties that will become useful later. Specifically, ReLU has an unbounded positive range and is very fast to compute. The derivative for ReLU is also well defined and simple (it is 1) for the entirety of the positive domain. Sigmoid, on the other hand, maps any input to the range between 0 and 1, and hence can be interpreted as a probability. It is more expensive to compute, since it involves exponentials and derivative tends to zero for (both positive and negatives) input values away sufficiently away from 0. These properties will be important both for both construction and optimization of the neural networks (though we will not go into details on this).

Interpretation of a neuron: It is compelling to ask what each neuron is computing. Looking at the functional form for the neuron, one can see that pre-activation map actually takes the form of a linear classifier that produces a real-valued score. An activation function then maps this score into a desired range. Therefoere a neuron can be thought of as a classifier or a feature extractor, that computes similarity (in the form of a dot-product) between the input and the weight vector.

1.2 Neural layers

We generally not going to work with a single neuron, but rather many many neurons working together both in parallel and in succession to produce a final output from the neural network. Hence instead of talking about neurons individually, it is more convenient to talk about them in terms of *layers*. A *layer* is a collection of neurons that share the same inputs but may have separate weight and bias parameters. As such they will produce different outputs (even though their inputs are identical). Importantly, neurons in a layer do not depend on one another and hence can be computed in parallel. In principle, any collection of differentiable functions that rely on the same inputs can be implemented as a layer of a neural network. However, we will restrict ourselves only to a few most common layer types in this class.

Once we have a layer or neurons, it becomes easier and more convenient to talk about their weights collectively as a matrix of the form `(number of inputs)` $\times$ `(number of neurons in the layer)` and bias as a vector of size `(number of neurons in the layer)`. Putting the neurons discussed above into a layer results in what is typically called a *fully connected* layer (or FC layer). The name *fully connected* refers to the fact that each neuron is connected to all the inputs. For example, if our input is a $32 \times 32 \times 3$ image, then each neuron in the fully connected layer will be connected to 3072 inputs. The number of parameters for such FC layer will be $(3072 + 1) \times$ `(number of neurons in the layer)`.

1.3 Building a Neural Network

A neural network is simply a collection of layers, where an output of one layer becomes the input for the next and so on and so forth until the final layer produces an output. In this way, any neural network can be characterized by a series of compositional functions. Note that while computations within the layer can be done in parallel, across layers the computations must be sequential, since one layer input depends on another layer output. Overall, a neural network typically has an *input* layer, which simply encompasses all the inputs (*e.g.*, most often in vision this is an image), a set of sequentially connected *hidden* layers and a single *output* layer. The output layer is no different than hidden layer, but is designed to produce a desired output in a particular form. The hidden layer is called hidden because it is simply not exposed to the user and does not produce any outputs that are consumed by the user.

For example, if we are building a neural network that will take CIFAR10 images of resolution $32 \times 32 \times 3$ and will classify them into 10 categories, then the input layer must be of size 3072 and the output layer must be of size 10. In addition, the output layer must output probability per class. This means that we generally would not be able to use a ReLU activation for the output layer, and instead need to use a Sigmoid to ensure the outputs are in the right range of 0 to 1 for probabilities. Note that the problem defines no constraints on the hidden layers inbetween; we can have many or few of them, and they can have ReLU or Sigmoid activation functions (though ReLU tends to work better in practice).

A neural network defined in this way will produce an output for any image $\mathbf{x}$ where the output will depend on the weights and biases of all the intermediate and the output layer. If we are able to set those weights and biases appropriately, then the network will output accurate predictions. In order for this to happen we need to optimize all of these parameters based on the objective or loss functions.

Generally, parameters of neural networks, which include neuron weights and biases (and sometimes parameters of activation functions themselves), are optimized using a gradient based optimization. The idea is simple, if we start at random weights and biases and then compute a gradient with respect to those parameters of the sum of loss functions over all training instances, we can use a *gradient descent* to descent to the local optimum of parameters that minimizes the loss over the entire training set. Specifically, we can iteratively update the weights and biases by taking a λ sized step into the negative direction of the gradient; λ is called a *learning rate*. The challenge of doing this is that this requires a pass over the entire training set to compute each gradient and take a single update step. Given that the training set may consists of

millions of examples, this becomes infeasible. Instead, we tend to use *stochastic gradient descent*, where we compute approximation to the gradient based on a random (small) mini-batch of samples. Because this tends to produce a noisier gradient, we often also need to reduce the learning rate λ to accommodate and ensure that we do not take too big of a step in a potentially wrong direction.

The key to the stochastic gradient descent is to compute the gradients for millions, and sometimes billions, of parameters in an efficient and accurate manner. This is typically done by a procedure called *backpropoagation*. Backpropagation is a form automatic differentiation which expresses a desired neural network as a computational graph where each node consists of an input, output or intermediate variable and edges correspond to mathematical operations. Traversing the computational graph in two passes allows us to compute all the necessary gradients. The *forward* pass simply computes the value of the function being represented by the neural network, *i.e.*, does inference. The *backward* pass computes all the necessary derivatives by traversing the graph in the reverse order, starting from the loss and towards the input image. The benefit of using auto-differentiation is that we can do symbolic differentiation at elementary function level, while at the same time avoiding overly complex symbolic derivative expressions by leveraging numeric simplification. In addition, computing gradients using a backwards traversal allows computation of gradients with respect to all parameters in a single pass, making this process very efficient. Additional details of this were covered in class.

2 Convolutional Layer

One issue with building a neural network out of Fully Connected (FC) layers is that the number of parameters quickly becomes infeasible when dealing with images. For example, when taking an image of resolution $224 \times 224 \times 3$ as input, each neuron in the first hidden FC layer would be required to have $150,525$ weights; a layer with 1000 neurons would require 150+ million weights. This is clearly not scalable.

Assuming each neuron is a classifier, or a feature extractor, we may be able to get away with these classifiers or feature extractors not looking at the full image, but rather on only the local patches (recall Bag of Words models). Conceptually, this can be achieved by making our neurons *specialize* on only a certain part of the input image. This is called a *Locally Connected Layer*. A locally connected layer is the one where each neuron is connected only to a specific $k \times k$ neighborhood of pixels within the input. For example, one neuron can be setup to be looking at a top-left corner of $k \times k$ pixels; another to a set of pixels shifted by one to the right and so on and so forth. Note that the number of neurons now would have to be a function of input resolution to ensure we look at all patches. The benefit of such a design that it reduces the number of parameters by a significant amount, since each neuron now has a number of parameters which is a function of the neighborhood (kernel) size and is not a function of input resolution.

However, a Locally Connected Layer also has some limitations. Mainly, (a) the number of parameters is still quite high and (b) we now have neurons that specialize on specific regions of the input. This seems to violate property of stationarity in images. A region of the image should be interpreted based on its content and not where it is located. In practice, this makes locally connected layers not particularly useful.

The solution to both issues outlined above is a *Convolutional Layer*, which is similar to a Locally Connected Layer in that it operates on $k \times k$ neighborhoods, but effectively re-uses the weight and bias for neurons being applied in different locations. An alternative interpretation of this layer is that it defines a neuron that operates on a $k \times k$ neighborhood and then applies it densely to the entire input by sliding it around. This way the computations resulting from a CNN neuron (often called a *filter*) is nearly identical to linear filtering with a convolution. The main difference is that the filter is non-linear owning to the activation function. As a result, each application of the filter results in a pre-activation map of depth 1. Applying K such filters will result in an output volume of depth M. Note that the filter, by convention, extends the full depth of the input volume. This means that for an input volume of size $W_i \times H_i \times D_i$ the

filter (or kernel) of size $k \times k$ would actually operate on the patches of size $k \times k$ and have $k \times k \times D_i$ weights and 1 bias, for the total of $k \times k \times D_i + 1$ learnable parameters. Additional parameters that may effect the behavior of this layer are *padding* and stride. Padding works identically to linear filtering, while stride generalizes convolution operation such that the shifting between application is set to a fixed constant S. Overall, given an input volume of size $W_i \times H_i \times D_i$ the resulting output feature volume after CNN layer would have a shape: $W_o \times H_o \times D_o$, where $W_o = (W_i - F + 2P)/S + 1$, $H_o = (H_i - F + 2P)/S + 1$ and $D_o = K$ (K here is the number of filters, F is integer padding, F is a filter size and S is the stride).

3 Pooling Layer

Another common layer type is a *Pooling Layer*, which is designed to add a certain degree of positional invariance while at the same time reducing spatial resolution of the feature maps. Similar to CNN layer, a pooling layer defines a kernel of size $k \times k$, however, in this case the kernel sizes are often even while in CNN layers they are often taken to be odd. With a neighborhood defined by a kernel the pooling layer summarizes information within this neighborhood by typically either taking a *max* or *average* over the corresponding values. In typical applications, the stride for pooling equals to the kernel size, which ensures that it operates on non-overlapping blocks of the image. Further, pooling operations are done per channel, meaning that the depth of the output volume will match the input, but spatial resolution is typically reduced in the process. For example, an input volume of $W_i \times H_i \times D_i$ after pooling will result in output volume $W_o \times H_o \times D_o$, where $W_o = W_i/F$, $H_o = H_i/F$ and $D_o = D_i$. Note that while the use of padding is possible with pooling as well, it is generally much less common.

4 Convolutional Neural Network

Convolutional neural networks are build by typically combining CNN, Pooling and Fully Connected layers in succession. The design often follows a typical paradigm of a few CNN layers, followed by a Pooling layer, followed by more CNN layers, more pooling and eventually a Fully Connected layer (or a couple of layers) that takes the final low-resolution feature map, vectorizes it (operation often referred to as *flattening*) and outputs probabilities per class. Often one proceeds from shallower to deeper layers, the resolution of the feature maps is reduced, while a channel dimensions increase.

5 NN Terminology

When talking about neural networks, it is best to know terminology adopted by the field.

Network structure or architecture refers to the structural design of the network. For example, number and types of layers used, forms of activations for those layers, connectivity of those layers, filter sizes where appropriate, and so on. The network structure is generally designed by a researcher or engineer and kept fixed. Defining the structure requires knowledge of the problem you are trying to solve and a lot of intuition about neural networks, how they perform inference and learn. A typical wisdom is to design a neural network using a core structure, often called a cell or a block, by iteratively stacking these. For example, ResNets are designed by stacking simple residual blocks, one on top of the other. Google's Inception architecture is a stack of Inception blocks. The general wisdom is that deeper architectures will perform better than shallower ones, but are often harder to optimize (due to vanishing and exploding gradient problems). Residual design gets around this by introducing residual connections, alleviating these problems to a large extent.

Loss or objective functions define how one should penalize predictions that do not match the ground truth. Common choices for such functions are Softmax + Cross Entropy, which models similarity between two distributions, and L1/L2 for regression problems. Note that, in general, the loss functions need to be differentiable since they are added to computational graph during learning and hence need to be backpropagated through. Design of the loss function can greatly impact both the performance and the learning of the neural network. Designing good loss functions requires both understanding of the problem being solved and the characteristics of the neural network and optimization involved.

Parameters in neural networks typically refer to the trainable parameters of the network. This includes all the weights and biases for linear, fully connected and CNN layers; potential parameters of activation functions. Parameters are what is being optimized during the learning of the neural network by stochastic gradient descent with back-propagation.

Hyper-parameters are meta-parameters that cannot be optimized directly, but rather need to be tuned "manually" or programmatically through discretized grid search. For example, hyper-parameters would include learning rate λ, batch size and others that we have not covered in class. The key is that these are the parameters with respect to which computing gradients is either difficult or completely impossible.