CS 194-129 Deep Neural Networks
Spring 2018 Canny Lecture 6

1 Reading

Architecting and Training Neural Networks

SOURCE: CS 231N

Commonly Used Activation Functions

An activation function (*nonlinearity*) takes a single number and performs a fixed operation on it.

- **Sigmoid** $\sigma(x) = 1/(1 + e^{-x})$: squashes a real-valued number into the range $[0, 1]$.
  - Large negative numbers become 0 and large positive numbers become 1.
  - Two main drawbacks: sigmoids saturate and kill gradients, and outputs are not centered at zero.
- **Tanh** $tanh(x) = 2\sigma(2x) - 1$: squashes a real-valued number into the range $[-1, 1]$.
  - Unlike the sigmoid neuron, its outputs are zero-centered.
  - Hence the tanh nonlinearity is always preferred to the sigmoid nonlinearity.
- **ReLU** $f(x) = \max(0, x)$: thresholds the activation at 0.
  - Greatly accelerates the convergence of SGD (potentially due to its linear, non-saturating form).
  - Does not involve expensive operations.
  - One main drawback: ReLU units can "die" (gradients forever zero) during training if the learning rate is too high.
- **Leaky ReLU** $f(x) = \mathbb{1}(x < 0)(\alpha x) + \mathbb{1}(x \geq 0)(x)$: thresholds negative values to *not quite* 0.
  - This is in an attempt to fix the "dying ReLU" problem.
  - The function output will be based on a small negative slope when $x < 0$.
- **Maxout** $\max(w_1^T x + b_1, w_2^T x + b_2)$: generalizes the ReLU and its leaky version.
  - Both ReLU and leaky ReLU are a special case of this form.
  - Enjoys all of the benefits of a ReLU unit without its drawbacks, but doubles the number of parameters for every single neuron.

For no fundamental reason, it is rare to mix and match different types of neurons in the same network. Typically, the go-to option is ReLU (with backup options being leaky ReLU and maxout).

Layer-wise Organization

In a **fully connected** layer (the most common layer type), neurons between two adjacent layers are fully pairwise connected, but neurons within a single layer share no connections.

When we say "N-layer neural network", we do not count the input layer. For example, a single-layer network describes one with no hidden layers (input directly mapped to output).

Generally, output layer neurons do not have an activation function. This is because the last output layer usually represents a quantity that is best modeled with arbitrary real-valued numbers.

Usually, we measure network size by number of parameters (and less frequently number of neurons).

Feedforward Computation

Feedforward computation typically consists of matrix multiplications interwoven with activation functions. One reason networks are organized into layers is so that they can be easily evaluated using matrix-vector ops.

The forward pass of a fully connected layer corresponds to one matrix multiplication followed by a bias offset and an activation function.

Representational Power

Neural networks with at least one hidden layer are *universal approximators.* They can (amazingly!) approximate any continuous function. However, this does not necessarily speak to empirical efficacy (i.e. we shouldn't necessarily use two-layer neural networks for everything).

It just so happens – i.e. it is an *empirical* observation – that neural networks work well in practice (justifications: they compactly express nice, smooth functions that fit well with the statistical properties of real data, and can be easily learned with known optimization algorithms). Nevertheless, they are far from the only approximators with such representational power.

Similarly, the fact that deeper networks can work better than two-layer networks is an *empirical observation*, not a result of them having any superior representational power.

Basically, representational power doesn't tell the whole story!

Complexity

In practice, three-layer FC nets often outperform two-layer FC nets, but going deeper rarely helps much more. By contrast, depth tends to be extremely important for CNNs (perhaps because images contain hierarchical structure and require multiple layers of processing).

Neural networks with more layers and neurons have a higher model capacity (i.e. they can express more complicated functions). However, while this allows the nets to understand more complicated data, it also increases the chance of **overfitting** (where models with high capacity fit the noise in the data instead of the assumed underlying relationship).

Smaller networks are harder to train with local (e.g. gradient descent) methods. Their loss functions have relatively few local minima, but many of these minima are easier to reach and are also bad. Conversely, bigger networks contain many more local minima which are much better in terms of their associated loss. Thus a smaller network creates more variance in the final loss (across different training sessions). With a larger network, all solutions are similarly good and we rely less on the luck of random initialization.

An important takeaway: we shouldn't use smaller networks just because we're afraid of overfitting. *Instead, we should use as big of a neural network as our computational budget allows, and then use regularization techniques to control overfitting.* Bigger is better!

Data Preprocessing

- **Mean subtraction:** subtracting the mean across every individual feature in the data (centers the cloud of data around the origin along every subtraction).
- **Normalization:** normalizing the data dimensions so that they are of approximately the same scale.
  - Apply this preprocessing if we have reason to believe that the different input features have different scales or units, yet should be of approximately equal importance to the learning algorithm.

  - This is not necessarily the case for images.
- **PCA and whitening:** performing PCA to reduce the dimensionality of the data (keeping the d dimensions of the data that contain the most variance).
  - This has the effect of decorrelating the data (making the covariance matrix diagonal).
  - We can also whiten the data, i.e. take the data in the eigenbasis and divide every dimension by the eigenvalue to normalize the scale. This transforms the data covariance matrix into the identity matrix.
  - Geometrically, whitening the data corresponds to stretching and squeezing the data into an isotropic Gaussian blob.
  - In practice, PCA and whitening are not used with convolutional networks.

Note that any preprocessing statistics (e.g. data mean) must only be computed on the *training* data, and then applied equally to all splits (train/val/test).

Generally, we'll center the data to have a mean of 0*, and normalize its scale to* $[-1, 1]$.

Weight Initialization

We can't initialize all of the weights to zero, because then every neuron would compute the same output and undergo the same gradient updates. Therefore, we usually initialize the weights to small random numbers and then normalize the variance of each neuron's output to 1 (or some constant). The normalization is done in order to prevent the variance of each neuron from growing with the number of inputs.

Batch normalization, which standardizes the linear output of each neuron, makes networks significantly more robust to bad initialization.

Regularization

L2 regularization penalizes the squared magnitude of all parameters (directly in the objective). This regularization scheme is partial to small, diffuse weight vectors ("use all weights a little, instead of some weights a lot"), and is sometimes referred to as "L2 weight decay."

L1 regularization causes the weight vectors to become sparse during optimization (lots of zeros).

Dropout is an extremely effective and simple form of regularization which works on top of other methods. It refers to setting a random subset of the neuron outputs to zero. Note that we need the expected outputs at test time to be the same as those at training time, so we scale the post-dropout outputs by $1/p$ during training time. (This is known as **inverted dropout**.)

Loss Functions

Classification losses include the SVM hinge loss (e.g. the Weston Watkins formulation) and the cross-entropy loss for softmax classifiers. Regression (predicting real-valued quantities) losses include the squared L2 norm and the L1 norm of the difference between predicted/actual values.

Note: it's harder to optimize L2 loss than cross-entropy loss, which is more stable and requires less fine-grained precision. (L2 is fragile to the point where dropout becomes questionable!) It is often preferable to try binning values, and to train a classifier on the bins instead of a regressor on the true values. It's also nice to have a classifier, because it gives us a distribution representing confidence in each class as opposed to a single real number.

2 Lecture

Recap

Each layer of a convolutional layer output corresponds to a single filter with a single set of weights.

CNN Examples

- **LeNet-5** was one of the first convolutional networks (with an architecture similar to that of today!), and was used in the context of digit classification.
- One major design flaw of **LeNet-5** was that it used sigmoids after every linear layer. This was done in order to keep the gradients from exploding (in that it squashed outputs to the range [-1, 1]). Unfortunately, it ended up causing vanishing gradients instead.
- **AlexNet**, which used a similar convolutional structure with ReLU activations (and some fancy normalization to prevent exploding gradients), was very successful on ImageNet and gained a lot of traction.
- **VGGNet** pushed the limits in terms of network depth; their deepest configuration had 19 convolutional fully connected layers.
- **GoogLeNet** (22 layers) had three outputs. Two were helper (auxiliary) losses to give more hints to the earlier, more shallow stages of the network. During training it would minimize the sum of all three losses.
- **GoogLeNet** included layers with 1×1 convolutions in order to reduce dimensionality.
- **ResNet** had a depth of 152 layers, along with groundbreaking accuracy on image classification benchmarks. In order to make this work, they added shortcut/residual connections to the network. These were simple identity connections, where the input x would be added to the linear output:

  $$H(x) = F(x) + x$$

  where $F(x)$ is the result of applying, say, a weight layer, a ReLU, and another weight layer on its input. The residual net adds the extra x at the end. *Motivation*: networks were learning a hierarchy of features up to some point, and the features were quite different, but beyond that point they were learning fine adjustments or exceptional cases to the same layer representations. In other words, the later layers were close to identity maps. Thus, in order to give the network a giant hint that it should represent the function it was learning as a giant identity map plus a delta, we add a residual signal coming all the way from the input. Then every layer receives a noisy version of the input. However, since the first layers learn the early representations extremely well, later layers learn to use that representation instead of the identity – and the end result is the acquisition of an excellent hierarchical description.
  - This is really just giving it good guidance at every stage of the learning.
  - It's similar to taking the same network and then training just the first layer, then training just the second, then training just the third... and so on. However, this is probably much faster.

If you use transfer learning, you don't always need an insane amount of data to train CNNs!

Summary

- CNNs typically stack CONV, ReLU, POOL, and FC layers.
- There is a trend toward smaller filters and deeper architectures.
- There is a trend toward getting rid of POOL/FC layers.
- There is a trend toward using only CONV-ReLU (fully conv layers), 1×1 convolutions, and softmax.

Activation Functions

An activation function is a unary function (a function of one value) that gives you a different value on the output. They include

- **sigmoid** functions (S-shaped functions) which typically start at 0 and saturate at 1,
- **hyperbolic tangent,** a two-sided saturating function (for negative and positive saturated values),
- **ReLU,** a rectifying unit, which tends to kill off gradients,
- **leaky ReLU,** which has a small negative slope for negative inputs,
- **maxout,** just the max of two weighted inputs, and
- **ELU,** a smooth version of ReLU and leaky ReLU.

Sigmoid functions (e.g. the logistic function) tend to kill gradients outside of the pseudo-linear zone in the middle. (It has a fast saturation outside of that region.) However, if we want to turn a real value into a binary value, this can be a good thing. For example, the control signals (forgetting or gating signals) in recurrent networks use this kind of element. In practice, though, sigmoid functions have largely been replaced by ReLUs.

If the input to a neuron is always positive, the gradient is always all positive or all negative. Hence there's only one way to go and the weights just have to keep increasing (or decreasing). This is why we want input data to be normalized, with mean zero.

Hyperbolic tangent does a symmetric clipping (in the range $[-1, 1]$ instead of $[0, 1]$). This makes a difference because it will operate linearly around zero instead of having a big offset at zero. It's used in recurrent networks to give us signed values that we can remember.

ReLU does not saturate in the forward direction, and allows modest growth in activations. Empirically it also trains quite a bit faster than sigmoid or hyperbolic tangent. However, it's not very good for approximating smooth functions (e.g. for a robot controller), and its output is not zero-centered. It also has the gradient-killing problem; a dead ReLU will never activate (and we will therefore never update again).

Leaky ReLUs do not saturate or die. They are guaranteed to never kill gradients. One might be defined $f(x) = \max(0.01x, x)$.

ELUs are smooth versions of ReLU and leaky ReLU. This is good for robot controllers. It also saturates the negative activations: we want the positive gradients to propagate all the way through, while negative gradients might be a little more remedial (though we still don't want them to get too big). Overall, ELUs both add smoothing and limit how much negative activation they can have. They also have outputs with mean close to zero.

Maxout *neurons* compose two layers of ReLU discontinuity by taking the max of two layers. They do not have the basic form of "dot product $\rightarrow$ nonlinearity," and are sometimes the most efficient unit to use if we are comparing two different embeddings of images (e.g. to compute a best joint embedding).

In practice, we should use ReLU on early image layers, and sigmoids for smooth functions (e.g. robot control) and logical functions AND/OR. We might also want to try out leaky ReLU, maxout, and ELU.

Weight Initialization

Xavier initialization: assume the input activations & weights are independent random values (so the variances of weights will multiply) with mean zero. If each output is created from fan_in inputs, the variance is fan_in times the variance of any one term. Since we want a variance of 1, we should scale the weight values by $1/\sqrt{fan\_in}$. In this way the energy on any input is approximately preserved on the output.

Note that ReLUs will kill off half the energy after every dot product. Thus, since the ReLU shrinks our output by a factor of $\sqrt{2}$, we should actually scale by $1/\sqrt{fan\_in/2}$ (we're effectively multiplying our weights by a factor of $\sqrt{2}$). This makes a surprisingly large difference.

Batch Normalization

Our networks do best when the activations are zero mean and unit standard deviation. (Sigmoids and tan functions behave differently if we have different scales of inputs.) In general, this also corresponds to getting rid of random bias in our inputs.

Therefore, we can normalize our activations. Note: we can either normalize on a per-activation basis (this is the default) or spatially. For "per-activation," we average one filter's output at one position across the images in a minibatch. The alternative, "spatially," averages across the entire plane of neurons with that filter in a minibatch. In practice, this tends to do better for images.

Note: batch normalization stops things from growing or vanishing. Also, there are differing opinions on whether to insert batch normalization before or after the nonlinearities. There are no rules, really – either may be acceptable.

References

[1] CS 231n neural network notes (pt. 1),
`http://cs231n.github.io/neural-networks-1/`

[2] CS 231n neural network notes (pt. 2),
`http://cs231n.github.io/neural-networks-2/`