Deep learning on Computational accelerators

Tutorial 4 : Training Neural Networks

Chaim Baskin, Alex Bronstein, Avi Mendelson, Aviv Rosenberg

Technion Israel Institute Of Technology,
Computer-Science department

November 11, 2018

Overview

Classification using NNs

Design principles

- Regularization

Optimizing neural networks

- Optimization challenges

Objective and Loss functions

A common objective is to *classify* correctly a set of examples, such that t_i is the class label. For each input x, the output of the network $y = F(x)$ indicates a probability measure over the set of classes C, and the classification is taken as

$$c = \arg\max_j y_j$$

where $y \in \mathbb{R}^{|C|}, \quad \sum_{j=1}^{|C|} y_j = 1, \quad 0 \leq y \leq 1$
In order to create this normalized measure, we use another non-linearity - *SoftMax* function

$$y_i = \frac{e^{x_i}}{\sum_{j=1}^{|C|} e^{x_j}}$$

which we can use for multi-class cross entropy regression.

Objective and Loss functions

The error function for cross-entropy regression is

$$E_{\text{cross-entropy}}(y,t) = -\sum_{j\in C} t_j \log y_j$$

and when considering t as an indicator, where c_t is the correct label

$$t = \begin{cases} 1, & \text{if} \quad j = c_t \\ 0, & \text{otherwise} \end{cases}$$

We get the "negative-log-likelihood" criterion

$$E_{\text{nll}}(y,t) = -\sum_{j\in C} t_j \log y_j = -\log y_{c_t}$$

Training NN - ”tricks of the trade”

Neural networks are usually *over-specified* - designed to have more parameters than the number of training samples

- This is usually a bad idea when considering a model in classical learning theory
- Turns out, this is crucial for training neural networks, where networks with larger number of weights are easier to train
- They are also easier to over-fit → regularization is needed

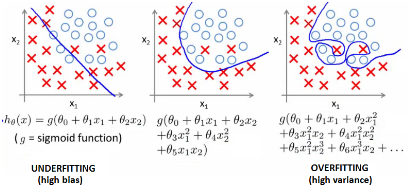
In some respects, it can be viewed as ”a blessing of dimensionality”, where a high dimensional weight space combat the input space's ”curse of dimensionality”.

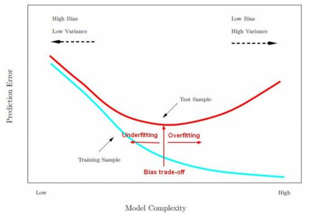
Training NN - overfitting

So the conventional wisdom for using neural networks as machine learning models turns out to be

- Design a neural network that is big enough - does not underfit on training data
- Regularize it so it will not overfit on validation data

Some reminders about overfitting:


x₂
x₁
$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$
(g = sigmoid function)
UNDERFITTING
(high bias)
x₂
x₁
$g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2 + \theta_5 x_1 x_2)$
x₂
x₁
$g(\theta_0 + \theta_1 x_1 + \theta_2 x_1^2 + \theta_3 x_1^2 x_2 + \theta_4 x_1^2 x_2^2 + \theta_5 x_1^2 x_2^3 + \theta_6 x_1^3 x_2 + \dots$
OVERFITTING
(high variance)



Regularization techniques

The challenge is now how to regularize our model.
Simplest methods for regularization are by modifying our training regime:

- Early stopping - This simply means to avoid overfitting by deciding to stop the training procedure according to a measured validation loss. Unregularized models tend to get worse at some point on a validation set, while still getting better on the training set.

Regularization techniques

- Data augmentation - introducing noise into our data samples, effectively enlarging it. Some kinds of noise have theoretical qualities such injecting Gaussian noise which regulate model complexity. Some are motivated by the task and data - such as affine transformations on images.

Regularization techniques

- Weight decay (L_2 regularization) - the loss function is added with a term that penalizes the L_2 of the weights

$$L(y,t;w) = E(y,t) + \eta\|w\|_2^2 \tag{1}$$

This is equivalent of decaying the weights in each gradient descent step. Causes the weights to remain small, and so fights over-fitting by smoothing the decision boundary

- Sparsity inducing regularization (L_1) - the loss function is added with a term that penalizes the L_1 of the weights

$$L(y,t;w) = E(y,t) + \zeta_1\|w\|_1 \tag{2}$$

or some hidden activations h within the network h

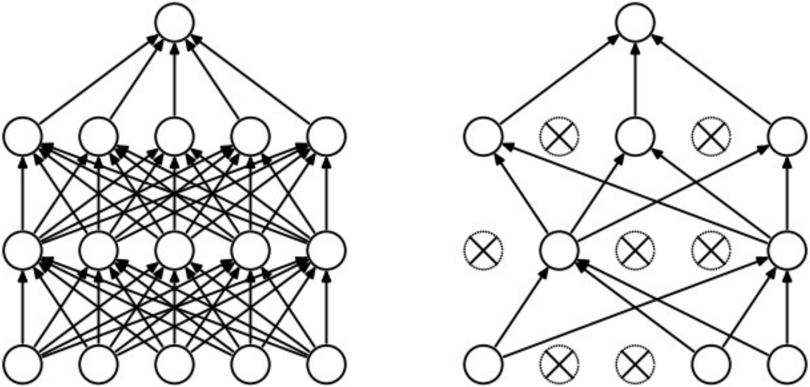
$$L(y,t,h;w) = E(y,t) + \zeta_2\|h\|_1 \tag{3}$$

Dropout

Dropout (Hinton '12) - perhaps the most useful regularization technique used in modern deep-learning models.
It consists of a noise-signal (usually a Bernoulli noise, but Gaussian noise was shown to work as well). The noise-signal is injected as a multiplication on intermediate hidden layers.

- For example, a Bernoulli dropout with $p = 0.5$ will zero half the activation of layer at each iteration.
- This noise will be removed when evaluating with a model, and hidden activation will be scaled by p.

Dropout



(a) Standard Neural Net

(b) After applying dropout.

Dropout

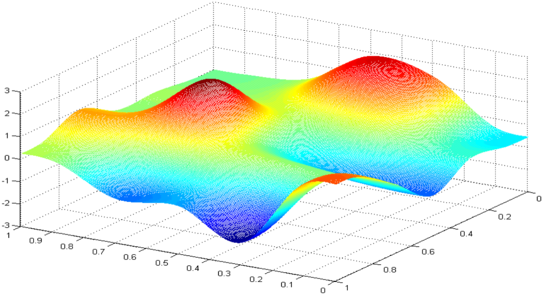
There are multiple explanation for the success of dropout:

- It encourages redundancy within a network - there can not be a high-dependency on a specific unit or set of units
- It can be seen as a model averaging technique - as 2^N different models are used under expectation when evaluating.
- Multiplicative noise (additive on gradients) - can allow better generalization, similar to benefits by data augmentations

Efficient optimization of NNs

Optimizing neural networks can be very challenging, as we're dealing with a highly non-convex problem, residing in a high dimensional space.

Problems can be observed even in a very simple non-convex error landscape:



Efficient optimization of NNs

Some obvious difficulties:

- Strong dependency on initial weights
- Possibly poor error estimation leading to noisy gradients
- Exploring a (very) high-dimensional space using only local information

We've seen the simplest update-rule:

naive SGD

$$w_{t+1} = w_t - \varepsilon \cdot \frac{\partial E}{\partial w_t}$$

where w is the optimized parameter (weight), E is the error (loss function estimation) and ε is the *Learning-Rate*

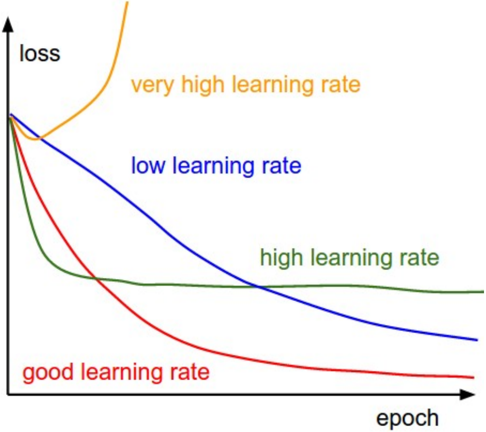
Annealing the learning rate

It is usually helpful to anneal the learning rate over time. Good intuition to have in mind is that with a high learning rate, the system contains too much kinetic energy and the parameter vector bounces around chaotically, unable to settle down into deeper, but narrower parts of the loss function.

There are three common types of implementing the learning rate decay:

- Step decay: Watch the validation error while training with a fixed learning rate. Reduce the learning rate by a constant whenever the validation error stops improving.
- Exponential decay: $\varepsilon_t = \eta^t \varepsilon_0 \qquad \eta < 1$
- Smooth decay: $\varepsilon_t = \frac{\varepsilon_0}{1+K\cdot t} \qquad K << 1$

Annealing the learning rate



Momentum

Momentum update is an approach that almost always enjoys better converge rates on deep networks.
This update can be motivated by viewing the trajectory on error surface as a ball rolling down a hill. We wish to accelerate towards the steepest slope direction, smoothing any irregularities in other directions.

SGD with momentum

$$v_{t+1} = \mu \cdot v_t - \varepsilon \frac{\partial E}{\partial w_t}$$

$$w_{t+1} = w_t + v_{t+1}$$

μ - *momentum* hyper-parameter. Usually around 0.9.

Per-parameter adaptive learning rate methods

All previous approaches we've discussed so far manipulated the learning rate globally and equally for all parameters.
Another type of optimizers allow adaptive learning rate for each one of the parameters. Common choices are:

- Adagrad: is an adaptive learning rate method that conditions each gradient element, with an estimate of its variance - making it inversely proportional to how noisy that direction is.
- RMSprop: adjusts the Adagrad method by using a moving average of squared gradients.
- Adam: The method computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients.

Per-parameter adaptive learning rate methods

Per-parameter optimization techniques have some nice benefits:

- Usually their performance is more robust with respect to their hyper-parameters - no need to check for best learning-rate, for example.
- They can greatly improve on convergence speed, especially in initial epochs/iterations

However, SGD+momentum with a carefully picked learning rate can usually converge to a better objective value. This is probably due to inherent noise which is helpful near the minima.

Optimizers Visualization

Optimization “tricks-of-the-trade”

Some heuristics you can use:

- Use data pre-processing - normalizing, and possibly *whitening* your data
- Shuffle data-samples each epoch
- Evaluate the ratio $\frac{\|\Delta W_t\|_2}{\|W_t\|_2}$ as a measure to efficient learning rate. Should be around 10^{-3}
- Disable momentum and turn to vanilla SGD when validation error completely flat-lines
- Lower regularization coefficient near convergence

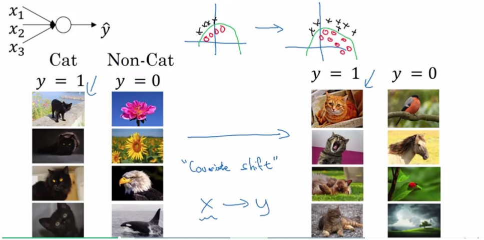
Batch Normalization

One of the most popular generalization techquies. How and why it works ?

- By normalizing the activations of each layer with statistics from the batch, we can use higher learning rates and get much faster convergence.
- Alleviates a lot of headaches with properly initializing neural networks by explicitly forcing the activations throughout a network to behave similar to a unit Gaussian.
- Possible because normalization is a simple differentiable operation.

Batch Normalization

- Batch normalization reduces the amount by what the hidden unit values shift around (covariance shift). To explain covariance shift, let's have a deep network on cat detection.



References

Some slides and figures were borrowed from Stanford's course on convolutional networks:

Course site