CS231n: CNNs for Visual Recognition **Spring 2017**

Lecture 6

Course Coordinator: Prof. Fei-Fei Li *Scribes: Akash Gupta*

Training NNs - Overview:

- One time setup - activation functions, preprocessing, weight initialization, regularization, gradient checking
- Training dynamics - babysit the learning process, parameter updates, hyperparam. optimization.
- Evaluation - model ensembles

Activation functions -

- Sigmoid - $\sigma(x) = \frac{1}{(1 + \exp -x)}$
  - Squashes numbers in $[0, 1]$
  - Problems:
    - Saturated neurons kill the gradient i.e. at high values of input gradient becomes 0.
    - Sigmoid outputs are not zero-centered. So gradient updates for further nodes can be inefficient. Consider e.g. of a neuron, if all the values of x are +ve - all gradients will be +ve, and if all x are -ve - gradients will be -ve. So this will restrict the gradients in specific directions and lead to delay in optimization.
    - exp() is computationally expensive.
- tanh -
  - Squashes numbers in $[-1, 1]$
  - zero centered
  - Problems:
    - Saturated neurons kill the gradient i.e. at high values of input, gradient becomes 0.
    - exp() is computationally expensive.
- ReLU - $f(x) = max(0, x)$
  - Does not saturate (in +ve region)
  - Very computationally efficient
  - Converges much faster than above two
  - More biologically plausible
  - Can initialize the inputs with a +ve bias to fire up initially.
  - Problems:
    - Not zero-centered output.
    - Annoying
- Leaky ReLU - $f(x) = max(\alpha x, x)$

  - All benefits of ReLU
  - Can initialize with a +ve bias to fire up initially.
  - Will not "die"

- ELU -

$$f(x) = \begin{cases} x & if x > 0 \\ \alpha(\exp x - 1) & if \leq 0 \end{cases}$$

  - All benefits of ReLU
  - Closer to zero mean outputs
  - -ve saturation regime adds some robustness
    * Computation requires exp().

- Maxout Neuron - Takes max of two linear functions

  - All benefits of ReLU
  - Will not saturate.
  - Will not "die".

Rule of thumb:

- Use ReLU (Careful with α)
- Try out Leaky/Maxout/ELU.
- Don't use Sigmoid

Data Preprocessing: Zero mean and standardize for good gradients and all features remain within one range. But in image processing and CV no need as already have a good distribution.

****In Images:** Center only. Two ways:

- Subtract the mean image.
- Subtract the per channel mean.

Q) *Does zero mean solve the sigmoid problem?*
A) For initial layers yes, but this problem increases as we go through deeper networks.

Weight initialization:

- If W=0, all neurons perform the same way.
- First idea: Small random numbers (gaussian with 0 mean and 1e-2 std.). Okay with small networks buts doesn't work with deeper networks - because activations will start diminishing as we go further in the layers which would instead affect the gradients by reducing their value as well due to the gradient being $X * downstream\_gradient$.
- So initialize W with a big value - Saturated neurons. Gradients again reduce to 0.

Solution:

1)Xavier initialization $->$ initializating every layer differently $->$ Basically sample from Gaussian distribution and scale by the number of inputs in each layer $->$ **Idea**: The variance of input should be same as the variance of output.

$$W = np.random.randn(fan\_in, fan\_out)/np.sqrt(fan\_in)$$

<u>Intuition</u> - If larger number of inputs we want smaller weights so divide by no. of inputs to maintain the variance in outputs, else, if smaller no. of inputs then we want larger weights.

<u>Drawbacks</u> - Doesn't work with ReLU as neurons die cuz approx half of the activations become 0. Change the formula by:

$$W = np.random.randn(fan\_in, fan\_out)/np.sqrt(fan\_in/2)$$

<u>Intuition</u> - Halving means adjusting to the fact that half the neurons gets killed so similar variance is seen in output.

Batch Normalization: Idea is to keep the activations in a gaussian range.

1. Compute the empirical mean and variance independently for each dimension.
2. Normalize: $\hat{x}^{(k)} = \dfrac{x^{(k)} - E[x^{(k)}]}{\sqrt{Var[x^{(k)}]}}$
3. Usually inserted after FC or Convolutional layers and before nonlinearity.
4. We can control this by: $y_i = \gamma \hat{x}_i + \beta$, network will learn that β is the mean and γ is the variance.

<u>Advantages</u> -

1. Improves the gradient flow.
2. Allows higher learning rates
3. Has some regularization effect since X is now computed using the empirical mean of all the activations in the batch adds some jitter to it. Maybe reduces the need for Dropout

**NOTE - At test time we don't compute the mean/std. Instead one fixed empirical mean of activations during training is used.

Babysit the learning process: Basically monitor training and adjust hyperparams.

1. Preprocess the data.
2. Choose the architecture.
3. Initialize the weights and other params.
4. Double check that the loss is reasonable without regularization and goes up after adding it.
5. Try to train with small portion of the training data and see if it overfits (loss = 0, accuracy = 1). Also, regularization should be turned off.

6. Start with small amount of regularization. (1e-6 too low lr - loss is barely changing, 1e6 too high lr - loss exploded and enountered NaNs)
7. Rough range of lr [1e-3,...,1e-5]

Hyperparameter Optimization:

1. Cross Validation Strategy - **First stage** - only a few epochs to get a rough idea. **Second stage** - longer runnning time, finer search.
2. Keep changing your ranges of lr according to the val. acc jumps.
3. Random search for choosing hyperparams.
4. Hyperparameters - network architecture, lr, it's decay schedule , update type and regularization(L2/Dropout)

****Some points to be noted**:

- If your loss curves are stabe at first and then suddenly start to decrease $->$ bad initialization.
- Big gap $->$ overfitting $->$ increase regularization str., no gap $->$ increase model capacity
- Check the $weight\_update/weight\_value$ (~ 0.001) as we don't want updates to be too high/low as compared to the values itself.