2025-08-14T00-36-34
Tags: DLS

DLS C2 week 3

- Tuning process
- Hyperparameter sampling scales
- Organizing the hyperparameter search process
- Batch norm in a single layer
- Batch norm into a neural network
- Why does batch norm work
- Batch norm at test time
- Softmax regression
- Deep learning frameworks
- One hot encoding

Tuning process

Selecting a good configuration of hyperparameters can be slow and inefficient without a structured approach.

Relative importance of hyperparameters that needs tuning:

1. Most important
   - learning rate α
2. Second-tier
   - momentum hyperparameter β (default of 0.9)
   - mini-batch size m_t
   - number of hidden units $n^{[l]}$
3. Third-tier
   - number of layers L
   - learning rate decay
4. Least important—rarely tuned
   - Adam's β_1, β_2, ε (defaults are 0.9, 0.999, 10^{-8} respectively)

Don't do grid search. Instead, do random search.

- Random search provides better coverage.
- Random search scales better with more hyperparameters.
- Grid search is inefficient when hyperparameters have unequal importance.

Coarse-to-fine search strategy:

1. Coarse search
   - Sample hyperparameters across a broad range to identify regions that yields promising metrics.

2. Fine search
    - Sample hyperparameters more densely in the identified regions.
    - Focus computational resources on refining best-performing areas.
    - Can be random.

Choose the configuration that maximizes train objectives, or performs best on dev set.

Hyperparameter sampling scales

Random sampling does not necessarily mean *uniformly random* over the range.

Suppose $\alpha \in [0.0001, 1]$:

- 90% of samples fall between 0.1 and 1.
- 10% fall between 0.0001 and 0.1.

Uniform linear sampling is heavily biased toward higher α. Many useful learning rates in the lower range get unexplored.

Sample on a log scale instead of a linear scale where each range gets equal attention when sampling.

Python implementation for logarithmic intervals $10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1$:

```
r = -4 * np.random.rand()
alpha = 10 ** r
```

General log-scale sampling steps between 10^a and 10^b:

1. $a = \log_{10}(\text{low value})$
2. $b = \log_{10}(\text{high value})$
3. Sample r uniformly in $[a, b]$
4. Set hyperparameter to 10^r

Suppose $\beta \in [0.9, 0.999]$:

- $\beta = 0.9$ averaging over last 10 values
- $\beta = 0.999$ averaging over last 1000 values

Instead, do $1 - \beta$:

- $\beta = 0.9 \Rightarrow 1 - \beta = 10^{-1}$
- $\beta = 0.999 \Rightarrow 1 - \beta = 10^{-3}$

Sampling previous β correctly:

1. Sample r uniformly in $[-3, -1]$
2. Set $1 - \beta = 10^r$

3. Tranpose $\beta = 1 - 10^r$

This makes equal exploration of 0.9-0.99 and 0.99-0.999 ranges. Why is this important? Sampling more densely near $\beta \approx 1$ is critical.

- $\beta = 0.9 \rightarrow 0.9005$ has minimal effect.
- $\beta = 0.999 \rightarrow 0.9995$ makes the average window leap ($1000 \rightarrow 2000$).

Organizing the hyperparameter search process

Ideas from one domain can transfer to others.

The optimal hyperparameters may become outdated due to:

- Change in data.
- Change in hardware.

Re-evaluate hyperparameters every few months to ensure their optimality.

Panda approach

Also known as: babysitting one model.

Use cases:

- When data is large.
- When computational resources are limited.
- When it is only possible to train a single or few models at a time.

Procedure:

1. Initialize model randomly and begin training.
2. Monitor performance metrics such as J and dataset error.
3. Adjust hyperparameter.
4. Keep on monitoring and make small changes.
    - better performance: adjust further.
    - worse performance: revert to previous model checkpoint.
5. Continue over days or weeks.

Advantages:

- Allows for fine-tuning of a single model.

Disadvantages:

- Time-consuming due to constant monitoring.
- May not explore a wide range of hyperparameters.

Caviar approach

Also known as: train many models in parallel.

Use cases:

- When computeational resources is abundant.
- When it is possible to train many models at once.

Procedure:

1. Launch multiple models with different hyperparameter settings.
2. Let it run for more than a day without constant intervention.
3. Compare resulting learning curves from cost function, error rates, and etc.
4. Pick the model with the best curve.

Advantages:

- Explores a wide range of hyperparameters efficiently.
- Less time-consuming.

Disadvantages:

- Wasted resources on poorly performing models.

Hybrid approach

Pandas can have multiple cubs in a single lifetime. It is similar for tuning a model.

Procedure:

1. Use panda approach for a model for weeks.
2. Start a new model and use panda approach again.

Advantages:

- Allows for wide exploration of hyperparameters while allowing fine-tuning.

Disadvantges:

- Careful planning of resources and time is critical.
- Complex.

Batch norm in a single layer

Steps for normalizing inputs in logistic regression or shallow models:

1. Subtract the mean (move the train set until $\mu = 0$).
   - $\mu = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}$
   - $x := x - \mu$
2. Normalize the variances.
   - $\sigma^2 = \frac{1}{m} \sum_{i=1}^{m} (x^{(i)})^2$ (element-wise squaring)
   - $x := x / \sigma$

Doing this will make elongated optimization contours into more spherical ones. Thus, making gradient descent converge faster.

Batch normalization applies normalization not only to the input layer features but also to intermediate hidden units.

- Normalize z (pre-activation) instead of a (post-activation)

Batch norm procedure

Given some intermediate values in a neural network layer, $z^{[l](i)} = z^{(i)} = z^{(1)}, \ldots, z^{(m)}$:

1. $\mu = \dfrac{1}{m} \sum_i z^{(i)}$
2. $\sigma^2 = \dfrac{1}{m} \sum_i (z^{(i)} - \mu)^2$
3. $z^{(i)}_{\text{norm}} = \dfrac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$ (normalize)
4. $\tilde{z}^{(i)} = \gamma \cdot z^{(i)}_{\text{norm}} + \beta$ (rescale)

γ and β are learnable parameters that are updated via back prop. The β here is not the momentum hyperparameter.

If $\gamma = \sqrt{\sigma^2 + \varepsilon}$ and $\beta = \mu$:

$$\tilde{z}_i = z_i$$

Thus, use z_i instead of $\tilde{z}_i$ before activating in the forward pass:

$$z^{[l](i)} \rightarrow \tilde{z}^{[l](i)} \rightarrow a^{[l](i)} = g^{[l]}(\tilde{z}^{[l]})$$

A key difference between the train input and hidden unit values: it is undesirable to force $\mu = 0$ and $\sigma^2 = 1$. Doing so, keep activations in a linear region.

That's why with γ and β makes the model adjust to:

- larger σ^2.
- a μ value different from 0.

This happens because $\gamma^{[l]}$ and $\beta^{[l]}$ set the variance and mean of $\tilde{z}^{[l]}$.

Batch norm into a neural network

Without batch norm:

$$x \xrightarrow{w^{[1]},b^{[1]}} z^{[1]} \rightarrow a^{[1]} = g^{[1]}(z^{[1]})$$
$$\xrightarrow{w^{[2]},b^{[2]}} z^{[2]} \rightarrow a^{[2]} = g^{[2]}(z^{[2]}) \rightarrow \dots$$

With batch norm:

$$x \xrightarrow{w^{[1]},b^{[1]}} z^{[1]} \underset{\text{batch norm}}{\xrightarrow{\beta^{[1]},\gamma^{[1]}}} \tilde{z}^{[1]} \rightarrow a^{[1]} = g^{[1]}(\tilde{z}^{[1]})$$
$$\xrightarrow{w^{[2]},b^{[2]}} z^{[2]} \underset{\text{batch norm}}{\xrightarrow{\beta^{[2]},\gamma^{[2]}}} \tilde{z}^{[2]} \rightarrow a^{[2]} \rightarrow \dots$$

The parameters are:

- $w^{[1]}, b^{[1]}, \dots w^{[l]}, b^{[l]}$
- $\beta^{[1]}, \gamma^{[1]}, \dots \beta^{[l]}, \gamma^{[l]}$

Example update of the new parameter:

$$\beta := \beta - \alpha \frac{\partial J}{\partial \beta}$$

Again, this $\beta^{[l]}$ is not related to the momentum and Adam hyperparameter β.

$\beta^{[l]}, \gamma^{[l]}$ are learnable and updated using the same optimizer as $w^{[l]}$ and $b^{[l]}$.

Batch normalization is often a one-line call when using DL frameworks like (PyTorch, Tensorflow, etc.).

Working with mini-batches

Batch norm is usually applied per mini-batches of the train set.

$$X^{\{1\}} \xrightarrow{w^{[1]},b^{[1]}} z^{[1]} \underset{\text{batch norm}}{\xrightarrow{\beta^{[1]},\gamma^{[1]}}} \tilde{z}^{[1]} \longrightarrow a^{[1]} = g^{[1]}(\tilde{z}^{[1]}) \xrightarrow{w^{[2]},b^{[2]}} z^{[2]} \longrightarrow \dots$$
$$X^{\{2\}} \xrightarrow{w^{[1]},b^{[1]}} z^{[1]} \underset{\text{batch norm}}{\xrightarrow{\beta^{[1]},\gamma^{[1]}}} \tilde{z}^{[1]} \longrightarrow a^{[1]} = g^{[1]}(\tilde{z}^{[1]}) \xrightarrow{w^{[2]},b^{[2]}} z^{[2]} \longrightarrow \dots$$
$$\dots$$
$$X^{\{t\}} \xrightarrow{w^{[1]},b^{[1]}} z^{[1]} \underset{\text{batch norm}}{\xrightarrow{\beta^{[1]},\gamma^{[1]}}} \tilde{z}^{[1]} \longrightarrow a^{[1]} = g^{[1]}(\tilde{z}^{[1]}) \xrightarrow{w^{[2]},b^{[2]}} z^{[2]} \longrightarrow \dots$$

Know that $z^{[l]} = w^{[l]}a^{[l-1]} + b^{[l]}$. With batch norm, each $z^{[l]}$ normalized as seen in step 3. By zeroing out the mean, any constant like b is cancelled out. Thus, the new parameterization is:

$$z^{[l]} = w^{[l]}a^{[l-1]}$$

$\beta^{[l]}$ from step 4 is now the parameter that controls the shift.

Parameter	Dimensions
$z^{[l]}$	$(n^{[l]}, 1)$
$\beta^{[l]}$	$(n^{[l]}, 1)$

Parameter	Dimensions
$\gamma^{[l]}$	$(n^{[l]}, 1)$

for $t = 1$ to num of mini batches.

1. Compute forward prop on $X^{\{t\}}$
   - Each hidden layer uses batch norm to replace $z^{[l]}$ with $\tilde{z}^{[l]}$.
2. Use backprop to compute $dw^{[l]}$, $d\beta^{[l]}$, $d\gamma^{[l]}$.
3. Update parameters with an optimizer (e.g., gradient descent)
   - $w^{[l]} := w^{[l]} - \alpha \cdot dw^{[l]}$
   - $\beta^{[l]} := \beta^{[l]} - \alpha \cdot d\beta^{[l]}$
   - $\gamma^{[l]} := \gamma^{[l]} - \alpha \cdot d\gamma^{[l]}$

Why does batch norm work

First intuition

Normalizing input features ($\mu = 0$, $\sigma^2 = 1$) prevents features from having different ranges. This stabilizes the scale of activations—thus, accelerating training.

Batch norm not only applies normalization to inputs but also to hidden unit values as well.

Second intuition

Weights in later layers (i.e., $l = 10$) can be sensitive to changes in earlier weights. Batch norm help reduce internal covariate shift.

Covariate shift:

- Given $X \to Y$ mapping, if the distribution of X changes, retraining is needed. This is true even if the ground truth from $X \to Y$ doesn't change.

Look at a certain hidden layer $l = 3$:

- It receives activations from the previous layer ($a_1^{[2]}$, $a_2^{[2]}$, $a_3^{[2]}$, $a_4^{[2]}$).
- From the view of $l = 3$, the activations are just features.

What is the problem?

- Parameters of earlier layers keep changing during training.
- The distribution of hidden unit values keeps shifting as earlier layers update. Therefore, later layers must constantly readapt. This leads to slows learning.

Batch norm solves this by making the mean and variance remain consistent across updates. It has the ff. effect:

- Weakens the coupling between early and later layers. This leads to easier optimization.
- Each layer can learn more independently of other layers—preventing destabilization of layer layers.

Third intuition

Batch norm has a slight regularization effect.

Each mini-batch x_t has the values $z^{[l]}$ scaled by the mean and variance by that one mini-batch.

- The mean and variance has little noise in it because it's computed just on the mini-batch.
- The process from $z^{[l]} \to \tilde{z}^{[l]}$ is also a little noisy—as it is computed using the slightly noisy mean and variance.
- It adds noise to each hidden layer's activations like dropout. However, batch noise adds lighter noise.

Mini-batch size tradeoff:

- Larger mini-batch: less noise, weaker regularization.
- Smaller mini-batch: more noise, stronger regularization.

Advices:

- Don’t treat batch norm as a regularizer. Use it mainly for normalization and speedup.
- If strong regularization is needed, still use dropout together with batch norm.

Batch norm handles data one mini-batch at a time.
At test time, you evaluate on a single example, not a batch. Thus, you can't compute batch stats at test time.

Batch norm at test time

During inference, you may process only one example at a time, not a mini-batch.

At test time, come up with a seperate estimate of μ and σ^2 using EWAs across the mini-batches.

Given some layer l:

1. Go through mini-batches $X^{\{1\}}, X^{\{2\}}, X^{\{3\}}, \dots$
2. Keep EWAs of $\mu^{\{1\}[l]}, \mu^{\{2\}[l]}, \mu^{\{3\}[l]}, \dots$
3. Keep EWAs of $\sigma^{2\{1\}[l]}, \sigma^{2\{2\}[l]}, \sigma^{2\{3\}[l]}, \dots$
4. Compute $z_{\text{norm}} = \dfrac{z - \mu}{\sqrt{\sigma^2 + \varepsilon}}$ using latest EWAs of μ and σ^2.
5. Compute $\tilde{z} = \gamma z_{\text{norm}} + \beta$

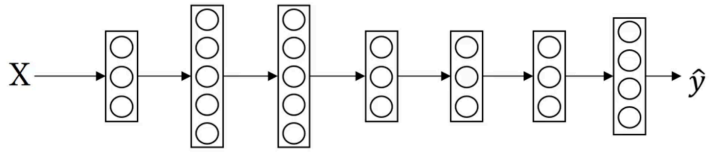
Softmax regression

Binary classification focuses on predicting one of two labels.

For multiple possible cases, use *Softmax regression*.

$$a^{[L]} = \frac{e^{Z^{[L]}}}{\sum_{i=1}^{4} t_i}$$

Notation	Meaning
C	num of classes



The network uses a *softmax layer* at the output:

- If $C = 4$, it indexes using $(0, \dots, 3)$.
- The output layer must have C units.
- Each output neuron corresponds to the probability of one class.
- $\hat{y}$ is a vector of shape $(C, 1)$.
- Each element of $\hat{y}$ is a probability of a class where all must add up to a value of 1.

In the final layer L, the activation for $z^{[L]} = w^{[L]}a^{[L-1]} + b^{[L]}$ is the *softmax* activation function:

1. Exponentiate logits (element-wise): $t = e^{z^{[L]}}$
2. Normalize classes: $a_i^{[L]} = \dfrac{t_i}{\sum_{j=1}^{C} t_i}$

Both (1) and (2) yields a $(C, 1)$ vector

Numerical example:

$$z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix}$$

$$t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} = \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix} \Rightarrow \sum_{j=1}^{C=4} t_j = 176.3$$

$$a^{[L]} = \frac{t}{176.3} = \begin{bmatrix} \frac{148.4}{176.3} = 0.842 \\ \frac{7.4}{176.3} = 0.042 \\ \frac{0.4}{176.3} = 0.002 \\ \frac{20.1}{176.3} = 0.114 \end{bmatrix} = \hat{y}$$

Softmax maps logits $z^{[L]}$to probabilities $a^{[L]}$.

Sigmoid, ReLU	• Takes a scalar, outputs a scalar	Softmax	• Takes a vector input and outputs a vector.

Generalization of logistic regression

Softmax regression generalizes logistic regression to C classes.

If $C = 2$, softmax reduces to sigmoid:

$$\hat{y}_0 = \frac{e^{z_0}}{e^{z_0} + e^{z_1}}$$
$$\hat{y}_1 = \frac{e^{z_1}}{e^{z_0} + e^{z_1}} = \frac{1}{1 + e^{-(z_1 - z_0)}} = \sigma(z_1 - z_0)$$

Decision boundaries:

Logistic regression	• 2 classes. • Decision boundary is linear. • One linear boundary separating two classes.
Softmax regression	• C classes. • Decision boundary is linear. • Multiple linear boundaries separating multiple classes.

A softmax layer without hidden layers can only form piecewise linear partitions of space. If you add hidden layers with nonlinear activations, the network can warp and curve the input space—creating nonlinear decision boundaries.

Training a softmax classifier

To train the network, define a cross-entropy loss function:

$$\mathcal{L}(\hat{y}, y) = -\sum_{j=1}^{C} y_j \log \hat{y}_j$$

Why?

- The loss is a maximum likelihood estimate where it increases the probability to the true class.
- The loss punishes low probability for the correct class.

Simplified gradient for softmax with cross-entropy:

$$dz^{[L]} = \hat{y} - y$$

Deep learning frameworks

Implementing deep learning algorithms in NumPy from scratch is important to gain intuition. However, as models get more complex—manual implementation is practical.

Examples of deep learning frameworks:

- TensorFlow
- PyTorch
- Keras

Criteria for choosing a framework:

- Ease of programming
- Training and running speed
- Open source with good governance

One hot encoding

Often times, you will have a Y vector with numbers ranging from $(0, C-1)$.

For example, if $C = 4$:

$$y = [1\ 2\ 3\ 0\ 2\ 1] \text{ is often converted to } \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Rows 1 through 4 correspond to classes 0, 1, 2, and 3, respectively.

This is *one hot* encoding. In the converted representation, exactly one element of each column is *hot*—set to 1.