Charlie Mackie
May 25th 2020

Improving Deep Neural Networks : Hyperparameter tuning / Optimization

Setting up ML app.

Data Split:

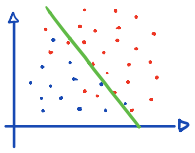
TRAIN	DEV	TEST
98 %	1 %	1 %

(*) will look different with less data, ex 100 examples total:
70 train, 30 dev/test

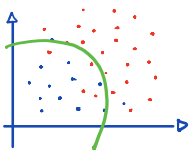
Distributions:

Test and dev. sets must come from the same distribution
(o.w. accuracy could be skewed poorly

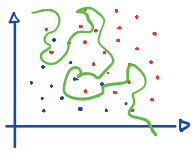
Bias / Variance:



High Bias
(underfitting)



Just Right



High Variance
(overfitting)

Data Set Accuracy Scenarios:

① Train Set Error: 1 % ⎫ high variance / overfit
Test Set Error: 11 % ⎭ does not generalize well

② Train Set Error: 15 % ⎫ high bias / underfit
Test Set Error: 16 % ⎭ error too far from Bayes

Human Error : How well would the most suitable human(s) perform
Bayes Optimal Error: approx. 0%.

Solutions: ① High Bias ⟶ • Try a bigger Network
• Try a different architecture

② High Variance ⟶ • Get more Data
• Regularization

Regularization

Goal: reduce overfitting / variance / accuracy difference

$$J(w,b) = \frac{1}{M}\sum_{1}^{M} \mathcal{L}(\hat{y}, y)$$

* add something to Cost function: this will change the gradient descent

L2 Regularization: $$J(w,b) = \frac{1}{M}\sum_{1}^{M} \mathcal{L}(\hat{y}, y) + \frac{\lambda}{2M}\|w\|_2^2$$

L1 Regularization: $$J(w,b) = \frac{1}{M}\sum_{1}^{M} \mathcal{L}(\hat{y}, y) + \frac{\lambda}{2M}\|w\|_1$$

* Now have λ as a hyperparameter for Dev. set tuning

In a Neural Network:

$$J(w,b) = \frac{1}{M}\sum_{1}^{M} \mathcal{L}(\hat{y}, y) + \frac{\lambda}{2M}\sum_{1}^{L}\|w\|_F^2$$

$$\|w\|_F^2 = \sum_{i=1}^{n^{(l-1)}}\sum_{j=1}^{n^{(l)}} \left(w_{ij}^{[l]}\right)^2$$ ← Frobenious Norm.

This process will effect Gradient descent...

$$w^{[l]} := w^{[l]} - \alpha\left[(dw)^{[l]} - \frac{\lambda}{m} w^{[l]}\right]$$ ((dw) from BP)

$$= w^{[l]} - \frac{\alpha\lambda}{m} w^{[l]} - \alpha\,(dw)^{[l]}$$ ((dw) from BP)

Why does this work?

- punishes weights with high value / variance
- pushes $w^{[l]}$ to be smaller ∴ closer to a linear slope in activation function

Dropout

Creates a probability parameter that represents if a node will be 'dropped' or not.

Creates more generalized / Lower variance models

Implementation : Inverted Dropout

(this could vary by layer)

(*) construct vector D3 and multiply D3 by A3, set Keep Prob Parameter

- d3 (dropout vector) = np.random.rand(A3.shape[0], x) < Keep Prob
- a3 = np.multiply(a3, d3)
- a3 /= keep prob ← need to retain general norm of weights

(*) Do not apply Dropout @ Test Time

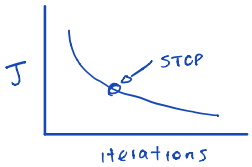
Why does Dropout work?

- Less reliance on a single feature (generalized)
- Smaller network from regularization

Other types of Regularization :

Data Augmentation → crop / rotate images to manipulate / increase the training set

Early Stopping →



Downsides of Some Regularization :

- can reduce orthoganalization (focus on individual tasks)
- want to focus on these seperately
  - optimize
  - generalize

Optimization

Normalize Inputs / Training Set

① Subtract mean : $\mu = \frac{1}{m} \sum x^{(i)}$, $x - \mu$

② Normalize variance : $\sigma^2 = \frac{1}{m} \sum (x^{(i)})^2$

(*) cost function is more symmetric and can step through with a higher learning rate

Vanishing / Exploding Gradients

weights being manipulated millions of times can cause them to reach extrem values

↳ use a different initialization : Glorlot / Xavier

$$W^L = np.random.randn(shape) * sqrt(2 / n^{[L-1]})$$

Gradient Checking :

$$\frac{f(\theta + \epsilon) - f(\theta - \epsilon)}{2\epsilon}$$

essentially random sample and check if the gradient you calculated (dW_L) is within 10^{-3} of above formula

Optmization Algorithms (make your algorithm faster)

Mini-batch Gradient Descent :

Train Data Set $X = \{ x^{(1)}, x^{(2)} \dots x^{(t)} \}$

↳ partition your data set and complete initialize → F prop. → J → B prop → update for each partition.

This speeds up the training process by performing gradient descent more frequently

⊛ Entire pass through Training Set (all mini batches) = Epoch

Choose mini-batch size between 1 — m :

① Batch Gradient descent : size = m

② Stochastic Gradient Descent (SGD) : size = 1

Typically powers of 2 are used in correspondance with Comp. memory

⊛ if m % batchsize != 0 then make the last batch smaller

Gradient Descent with Momentum

uses Exponentially weighted Averages: (similar to SMA)

$$V_T = \beta(V_{t-1}) + (1-\beta)\Theta_t$$

$\frac{1}{1-\beta}$ = number of x (time intervals) average is calculated over

* notice how its recursive, this value accounted for all previous values in β range

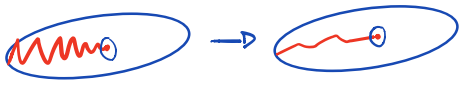
For GD with mom., change the update step:

- calculate $Vdw = \beta Vdw + (1-\beta)dW$

  then $w = w - \alpha Vdw$

  ↑ exponentially weighted average

reduces noise:



* Typically use $\beta = 0.9$, smaller β = smaller average period

RMS Prop (Reduces vertical movement / Noise)

$$w := w - \alpha \frac{dw}{\sqrt{Sdw}} \qquad b := b - \alpha \frac{db}{\sqrt{Sdb}}$$

* where $Sdw = \beta Sdw + (1-\beta)dW^2$
  $Sdb = \beta Sdb + (1-\beta)db^2$ ← smaller and reduces oscillations in G.D.

* Purpose: use a higher learning rate = faster

Adam Optimization: RMS Prop + Momentum (Adaptive Movement Estimation)

$$Vdw = \beta_1(Vdw) + (1-\beta)dw$$
$$Sdw = \beta_2(Sdw) + (1-\beta)dW^2$$

} Bias' are also corrected

$$w := w - \alpha \frac{V dw}{\sqrt{S dw} + \epsilon} \qquad b := b - \alpha \frac{V db}{\sqrt{Sdb} + \epsilon}$$

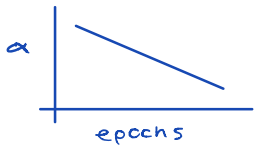
↑ avoid NaN division

hyperparameters ⟶ $\epsilon = 10^{-8}$, α = LR, $\beta = 0.9$, $\beta_2 = 0.99$

Learning Rate Decay — slowly reduce LR through Epochs

Intuition: take smaller steps when approaching convergence

$$\alpha = \frac{\alpha_0}{(1 + (\text{decay rate})(\text{epoch Num})}$$



Hyperparameter Tuning

Things to tune:
- α (learning rate)
- hidden units
- Layers
- β_1 and β_2 (Adam)
- mini-batch size
- α decay

(*) should tune these on a Logarithmic Scale

↳ cant just periodically iterate; missing lots of possible results

want to sample more densely at smaller/more precise values

Batch Normalization (accelerates training)

has a regularizing effect also...

$$\mu = \frac{1}{m}\sum x^{(i)}, \quad \sigma^2 = \frac{1}{m}\sum (x^{(i)} - \mu)^2$$

$$x_{norm} = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

⟶ expressed as: $\tilde{z}_i = \gamma z_i + \beta$ where γ and β are params. that we can learn.

(*) this is performed within the context of each mini-batch

Benefits:
- input values are more stable while later layers train
- more independance between layers

Softmax Regression (for multi-class classification)

output a vector of probabilities that sum to 1

$$z_L = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \qquad a_L = \begin{bmatrix} e^1 / (e^1 + e^2 + e^3) \\ e^2 / \dots \end{bmatrix} \qquad a_L = \frac{e^{(z)}}{\sum_i e^{(z_i)}}$$

could have a "hard-max" where max is one-hot

$$\text{loss function} = \mathcal{L}(\hat{y}, y) = -\sum y_j \log(\hat{y}_j)$$