

Lecture 04. Backpropagation and computation graphs

I. Matrix gradients for our simple neural net and some tips

1. Derivative wrt a weight matrix

Neural network과정은 아래와 같이 진행된다.

$$s = \mathbf{u}^T \mathbf{h}$$

$$\mathbf{h} = f(\mathbf{z})$$

$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

1) **FeedForward** : 벡터 \mathbf{x} 가 weight matrix와 곱해져서 output vector를 만들어진다.

2) **BackPropagation** : output vector를 weight matrix에 대해 미분한다. FeedForward 과정을 통해 나온 예측 값과 실제 값을 통해 error signal vector를 만들고, chain rule을 이용해 가중치를 업데이트 한다.

$$\frac{\partial s}{\partial \mathbf{W}} = \frac{\partial s}{\partial \mathbf{h}} \frac{\partial \mathbf{h}}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \mathbf{W}} \quad (\text{chain rule})$$

2. Deriving gradients for backprop

$$\frac{\partial s}{\partial \mathbf{W}} = \boldsymbol{\delta} \frac{\partial \mathbf{z}}{\partial \mathbf{W}} = \boldsymbol{\delta} \frac{\partial}{\partial \mathbf{W}} \mathbf{W}\mathbf{x} + \mathbf{b}$$

-변수를 잘 정의하고 차원을 계속 생각하고 있다.

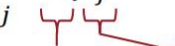
-Chain rule을 숙지한다.

-model의 마지막에서 softmax를 계산할 때 correct class/incorrect class를 따로 계산한다.

-행렬 미분이 헛갈리면 성분 별 미분을 연습한다.

-shape convention : hidden layer에 도착하는 δ 의 차원=해당 hidden layer의 차원

$$\frac{\partial s}{\partial W_{ij}} = \delta_i x_j$$


Error signal from above Local gradient signal

3. Deriving gradients wrt words for window model

-window에 등장한 단어들이 update되고 단어 벡터들이 task에 더 도움이 되게끔 변화하는 방식으로 x 각각의 window 값은 gradient를 받는다.

$$\nabla_x J = W^T \delta = \delta_{x_{window}} \quad x_{window} = [x_{museums} \quad x_{in} \quad x_{Paris} \quad x_{are} \quad x_{amazing}]$$

$$\delta_{window} = \begin{bmatrix} \nabla_{x_{museums}} \\ \nabla_{x_{in}} \\ \nabla_{x_{Paris}} \\ \nabla_{x_{are}} \\ \nabla_{x_{amazing}} \end{bmatrix} \in \mathbb{R}^{5d}$$

-**downstream task** : 향후 수행할 task(POS(품사판별), NER(개체명 인식)에 맞추어 가중치 업데이트를 진행하는 것.

4. A pitfall when retraining word vectors

-데이터를 함부로 retraining 시키는 것은 좋지 않다. Retraining 시키게 되면 가지고 있는 데이터에 따라 이 세단어의 임베딩 벡터가 달라질 수 있기 때문이다.

-pre-trained : 이미 사전에 학습되어 있는 것을 가져다가 쓰는 것. 대규모 말뭉치 & 컴퓨팅 파워를 통해 embedding 값을 만든다. 해당 embedding에는 말뭉치의 의미적, 문법적 맥락이 모두 포함되어 있다. 강의에서 웬만하면 pre-trained word vector를 이용하도록 권유.

-fine-tuning : pre-trained된 모델을 활용해 task에 맞게 조금 고쳐서 학습시키는 것. 이후 Embedding을 입력으로 하는 새로운 딥러닝 모델을 만든다. 풀고 싶은 구체적인 task와 가지고 있는 소규모 데이터에 맞추어 embedding을 포함한 모델 전체를 update 한다.

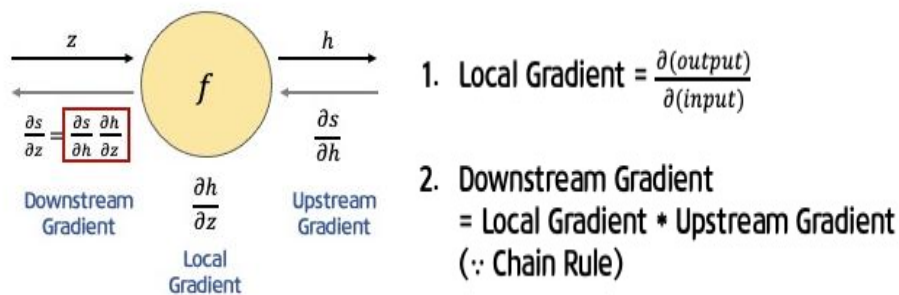
-retraining : data를 새로 가져와서 train 과정부터 다시 학습시키는 것. 100만 개 이상의 데이터가 있다면 retraining이 성능에 도움이 될 것이다.

5. Fine-tuning 진행 방법 결정

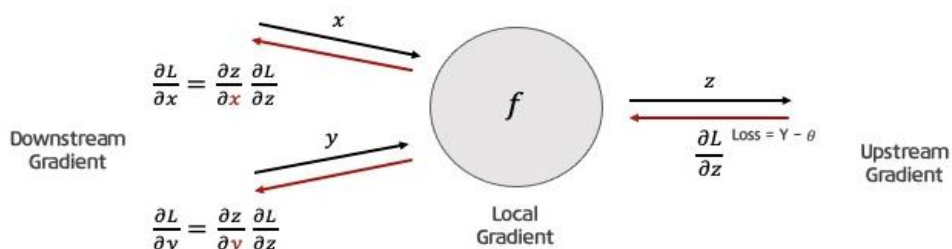
- 1) 크기가 크고 유사성이 낮은 데이터 : 데이터가 크므로 내가 원하는 대로 다시 학습.
- 2) 크기가 크고 유사성이 높은 데이터 : classifier와 모델의 마지막 단 계층 일부만 학습시켜도 충분.
- 3) 크기가 작고 유사성이 낮은 데이터 : 너무 많은 계층을 새로 학습시키면 underfitting.
너무 적은 계층을 학습시키면 제대로 학습되지 않을 것이다. **Data augmentation**
- 4) 크기가 작고 유사성이 높은 데이터 : 새로운 classifier만 잘 만들어서 학습시킨다.

II. Computation graphs and backpropagation

1. Forward & Back propagation



역전파 분해



2. Computational Graph

- Forward : Topological sort로 정렬한 뒤 노드를 지남.(사이클이 없는 방향 그래프의 노드들이 방향성을 거스르지 않도록 노드 정렬)
- Backward : output node에서 1로 gradient 시작. Topological sort의 반대 순서로 노드를 지나며 이어지는 노드에 대해 local gradient 계산.

III. Stuff you should know

Backpropagation을 그냥 진행하게 되면 아래와 같은 문제가 발생한다.

-Vanishing Gradient : sigmoid 함수의 미분값의 범위가 (0, 1/4)가되어 작은 값이 계속적으로 곱해지는 것이기 때문에 gradient 값이 0으로 해결한다.

→ Activation function, weight initialization으로 해결

-Dying RELU : input으로 음수가 들어오게 되면 backpropagation 진행 시 0값이 곱해지게 되어 사라져 버린다,

→ Batch normalization, optimization으로 해결

Overfitting : 가지고 있는 데이터에 지나치게 맞추어 학습하여 새로운 데이터셋에 대해 제대로 예측하지 못하게 된다.

→ Regularization, Dropout으로 해결

1. Regularization to prevent overfitting

$$J(\theta) = \frac{1}{N} \sum_{i=1}^N -\log \left(\frac{e^{f_{y_i}}}{\sum_{c=1}^C e^{f_c}} \right) + \lambda \sum_k \theta_k^2$$

-loss를 구할 때 LSE(Least Squared Error)가 최소가 되는 값, 즉 unbiased estimator를 찾는 것이 이상적이지만 이를 위해서는 parameter가 증가할 수 밖에 없으므로 모델이 복잡해짐. 목적함수에 항을 하나 더 추가하여 모델이 복잡해지도록 penalty를 부여하여 biased 하지만 smaller estimator인 loss 값을 찾는 것이 regularization이다.

2. Vectorization

-word vector를 개별적을 실행하는 것보다 한 개의 matrix를 만들어서 실행하는 것이 빠르며, 이 과정을 vectorization이라고 한다.

3. Activation Function

-non-linearity : ReLU를 제일 먼저 고려(간단하면서 성능 좋음).

-ReLU를 조금씩 변형시킨 activation function등 다양한 활성화함수가 등장하고 있음.

4. Initialization

- neural network 학습 전 가중치들을 초기화하는 과정. Gradient vanishing/Exploding 방지
- 일반적으로 작은 random value로 parameter의 초기 값을 줘야함. (hidden layer의 bias term은 0으로 초기값 부여. 다른 weight는 uniform(-r, r)에서 sampling)
- Xavier initialization : n_{in} , n_{out} 에 맞게 weight variance 조정

$$\text{Var}(W_i) = \frac{2}{n_{in} + n_{out}}$$

5. Optimizers

- 목적함수 f의 최솟값을 찾는 알고리즘.
- 보통 SGD를 사용해도 최적화가 좋지만 더 좋은 값을 얻기 위해서는 learning rate 튜닝.
- 복잡한 neural net 구조에서는 'adaptive' optimizer가 성능이 좋음.
- Adam(Adaptive Optimizer) : 상대적인 업데이트 양에 따라 step size 조정
- RAdam(Rectified Optimizer) : adaptive learning rate term의 방식을 rectify 함으로써, 학습의 안정성 확보
- Adagrad, RMSprop

6. Learning rate

- 0.01 정도의 일정한 learning rate를 일반적으로 사용. 보통 10배수의 배수로 하고 너무 작으면 학습이 느리고 너무 크면 발산한다.
- 학습이 진행될수록 learning rate를 감소시키는 방법 : k epoch 마다 반으로 줄이기.

Cycling learning rate(cyclic으로 주기를 만드는 방법)

$$lr = lr_0 e^{-kt}$$