

Hands-On Machine Learning with Scikit-Learn & TensorFlow

Chapter 11. Training Deep Neural Nets

181120 Dongwoo Kim



Software Safety
Engineering LAB

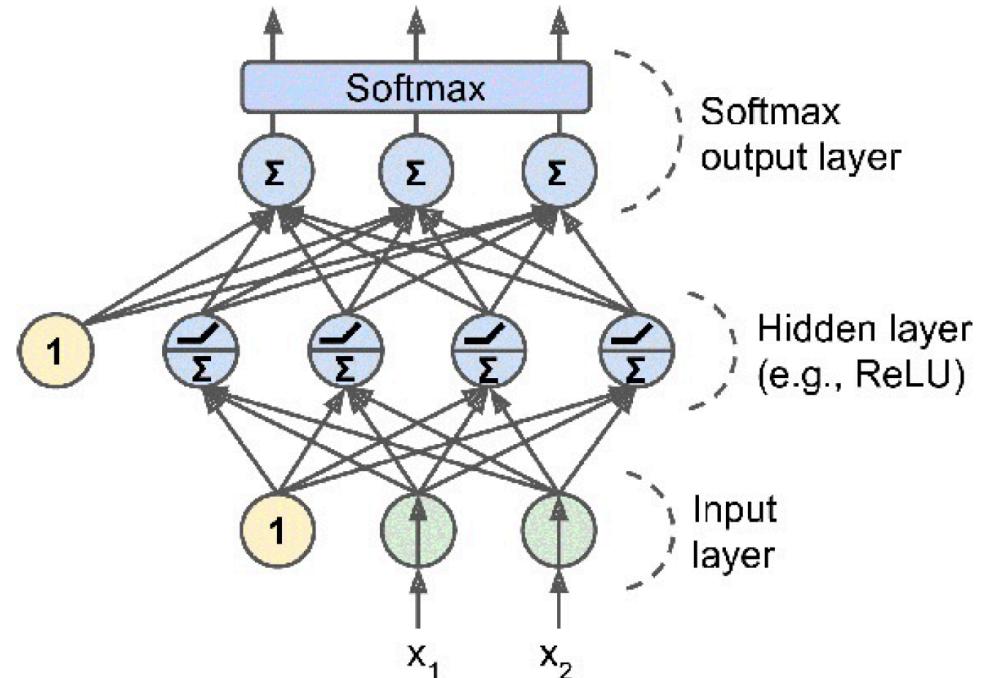
Overview

- 심층 신경망은 두 개 이상의 hidden layer가 있는 신경망
- 심층 신경망을 이용하여 복잡한 문제를 더욱 효과적으로 다루기 위한 방법을 본 장에서 소개
 - 활성화 함수의 종류와 특징 / 배치 정규화를 통한 그레디언트 소실과 폭주 문제 감쇄
 - 미리 훈련된 층 재사용하기 / Faster Optimizers / 과대 적합을 피하기 위한 규제 방안
 - 실용적 가이드라인에 관한 소개

Background

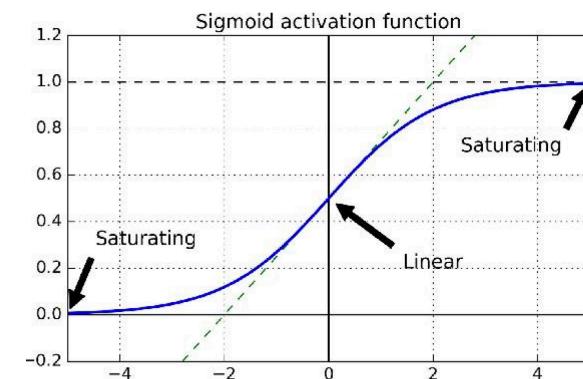
- 심층 신경망을 다룰 때 발생하는 문제

1. 그레디언트 소실/폭주 문제에 직면할 수 있다.
2. 훈련이 극단적으로 느려질 수 있다.
3. 과대적합될 위험이 높다.



그래디언트 소실/폭주 문제

- 역전파 알고리즘 적용 과정에서 출력층에서 입력층으로 오차 그레디언트를 전파하고 경사 하강법 적용 시 하위층으로 진행될 수록 그레디언트는 점점 작아지게되어 하위층은 수정되지 않을 수 있음 (반대의 경우에는 과하게 변화)
- 배경 : 로지스틱 활성화 함수와 무작위 초기화를 사용한 심층 신경망 학습
- 두 방법을 사용하여 심층 신경망 학습 시 매 층을 지날 때마다 출력의 분산이 커져 가장 높은 층에서는 0이나 1로 수렴



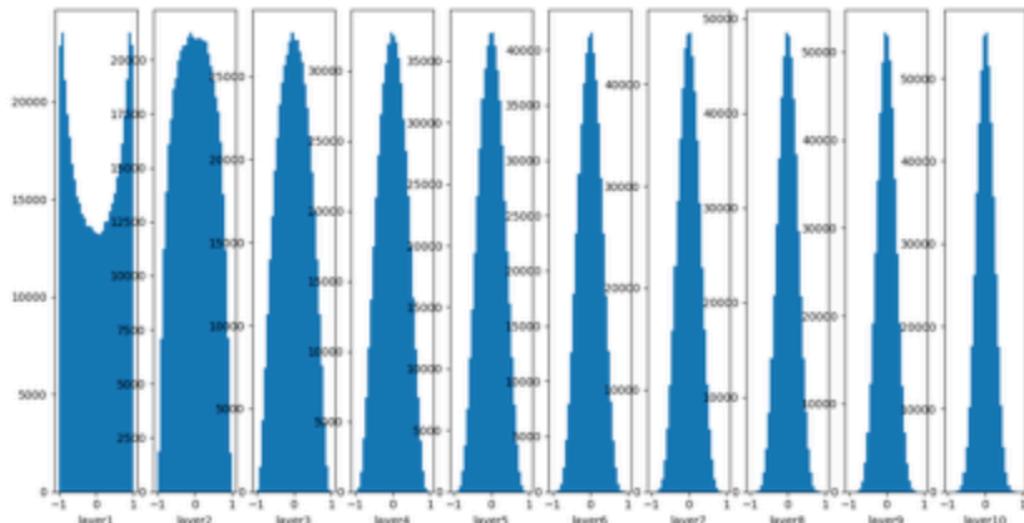
세이비어 초기화와 He 초기화

- 각 층의 출력에 대한 분산이 입력에 대한 분산과 같아야 하며 역방향으로 층을 통과하기 전후의 그래디언트 분산이 동일해야 한다.

Activation function	Uniform distribution $[-r, r]$	Normal distribution
Logistic	$r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
Hyperbolic tangent	$r = 4\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = 4\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
ReLU (and its variants)	$r = \sqrt{2}\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

* n_{input} 과 n_{output} 은 층의 입력과 출력 연결 개수

* 기존 무작위 초기화 방법에서는 평균 0, 표준편차 1을 사용했었음



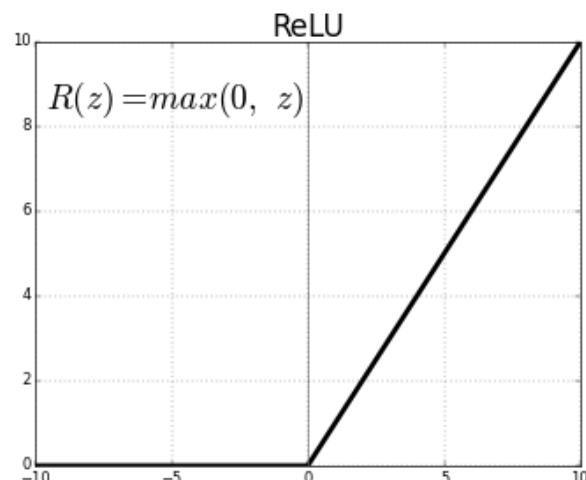
<세이비어 초기화를 적용한 후 분포가 갈수록 고르게 변하는 모습>

- 활성화 함수로 ReLU 등을 사용할 때에는 He 초기화를 사용

```
51      # normal distribution
52      tf.contrib.layers.variance_scaling_initializer()
53      hidden1 = tf.layers.dense(X, n_hidden1, name="hidden1", activation=tf.nn.relu,
54          kernel_initializer=he_init, name="hidden1");
```

활성화 함수

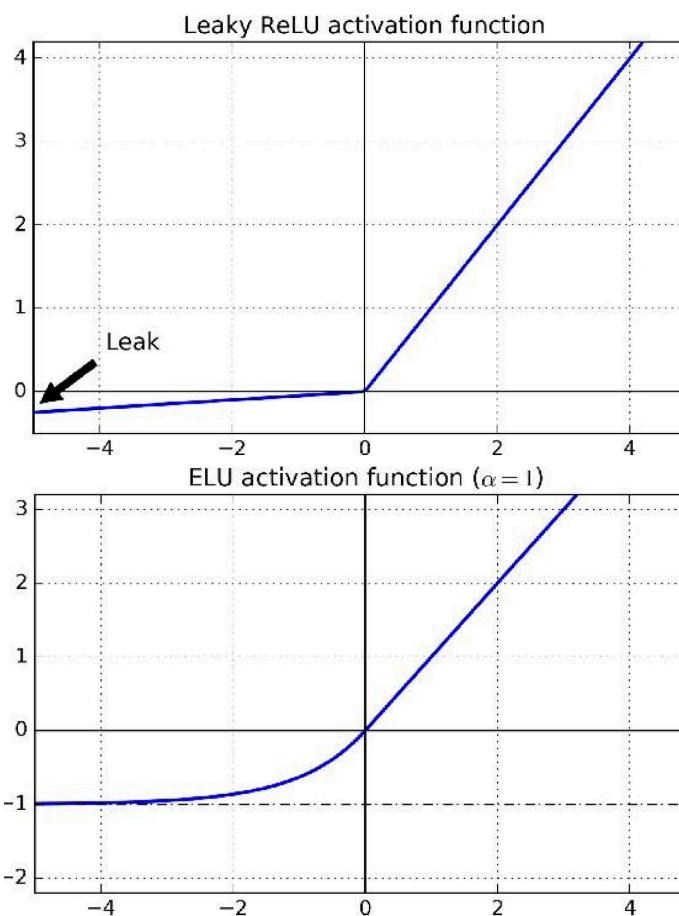
- 활성화 함수를 잘못 선택할 경우 그래디언트 소실/폭주 문제에 직면할 수 있다.
- 기존에는 시그모이드 함수가 뉴런의 방식과 비슷하여 최선이라 생각하였지만 다른 활성화 함수가 훨씬 더 잘 작동



$$\text{ReLU}(z) = \max(0, z)$$

특정 양수값에 수렴하지 않으며 연산도 빠름

문제: 가중치 합이 음수가되면 항상 0을 출력



$$\text{LeakyReLU}(z) = \max(az, z)$$

음수 값이 입력되더라도 다시 깨어날 가능성 존재

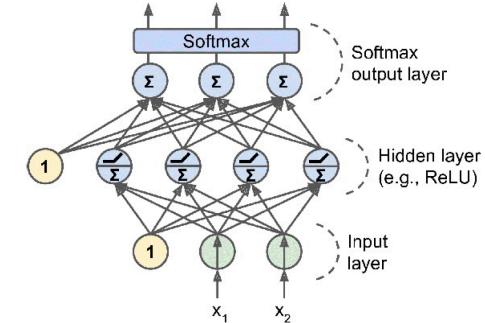
변종으로 RReLU와 PReLU가 존재

$$\text{ELU}_\alpha(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$$

ReLU보다 적은 훈련 시간과 테스트 성능도 높은
빠른 훈련은 가능하지만 지수함수 연산을 수행하기
때문에 테스트시에는 더 느리다.

배치 정규화

- 활성화 함수 교체로 그래디언트 소실/폭주 문제는 크게 줄일 수 있지만 훈련 과정 중 이전 층의 파라미터가 변함에 따라 다음 층의 입력 분포가 달라질 수 있다.
- 본 과정에서는 활성화 함수를 통과하기 전에 1) 입력 데이터의 평균을 0으로 만들고 2) 정규화하며 3) 두 개의 새로운 파라미터를 이용하여 입력 데이터의 최적 스케일과 평균을 학습한다.



```
42 # 2.2. Pre-defined & standard way to make layers - also, fully connected
43 with tf.name_scope("dnn"):
44     # 2.2.1. in case, standard distribution
45     hidden1 = tf.layers.dense(X, n_hidden1, name="hidden1", activation=tf.nn.relu);
46     hidden2 = tf.layers.dense(hidden1, n_hidden2, name="hidden2", activation=tf.nn.relu);
47     logits = tf.layers.dense(hidden2, n_outputs, name="outputs");
```

// 3개의 layers를 정의



```
42 # 2.2. Pre-defined & standard way to make layers - also, fully connected
43 with tf.name_scope("dnn"):
44     # 2.2.1. in case, standard distribution
45     hidden1 = tf.layers.dense(X, n_hidden1, name="hidden1");
46     bn1 = tf.layers.batch_normalization(hidden1, training=training, momentum=0.9)
47     bn1_act = tf.nn.elu(bn1)
48     hidden2 = tf.layers.dense(bn1_act, n_hidden2, name="hidden2", activation=tf.nn.relu);
49     bn2 = tf.layers.batch_normalization(hidden2, training=training, momentum=0.9)
50     bn2_act = tf.nn.elu(bn2)
51     logits_before_bn = tf.layers.dense(bn2_act, n_outputs, name="outputs");
52     logits = tf.layers.batch_normalization(logits_before_bn, training=training, momentum=0.9);
```

// 1. 각 Layer를 정의
// 2. 배치 정규화를 수행
// 3. 활성화 함수를 수행

배치 정규화 (2)

- 중복 코드를 partial 함수로 template을 만들어 처리

```
42 # 2.2. Pre-defined & standard way to make layers - also, fully connected
43 with tf.name_scope("dnn"):
44     # 2.2.1. in case, standard distribution
45     hidden1 = tf.layers.dense(X, n_hidden1, name="hidden1");
46     bn1 = tf.layers.batch_normalization(hidden1, training=training, momentum=0.9) // 정규화 layer를 세 번 정의
47     bn1_act = tf.nn.elu(bn1)
48     hidden2 = tf.layers.dense(bn1_act, n_hidden2, name="hidden2", activation=tf.nn.relu);
49     bn2 = tf.layers.batch_normalization(hidden2, training=training, momentum=0.9)
50     bn2_act = tf.nn.elu(bn2)
51     logits_before_bn = tf.layers.dense(bn2_act, n_outputs, name="outputs");
52     logits = tf.layers.batch_normalization(logits_before_bn, training=training, momentum=0.9);
```



```
43 # 2.2. Pre-defined & standard way to make layers - also, fully connected
44 with tf.name_scope("dnn"):
45     # 2.2.1. in case, standard distribution
46     my_batch_norm_layer = partial(tf.layers.batch_normalization, training=training, momentum=0.9) // 같은 template을 이용하여
47     hidden1 = tf.layers.dense(X, n_hidden1, name="hidden1");
48     bn1 = my_batch_norm_layer(hidden1)
49     bn1_act = tf.nn.elu(bn1)
50     hidden2 = tf.layers.dense(bn1_act, n_hidden2, name="hidden2", activation=tf.nn.relu);
51     bn2 = my_batch_norm_layer(hidden2)
52     bn2_act = tf.nn.elu(bn2)
53     logits_before_bn = tf.layers.dense(bn2_act, n_outputs, name="outputs");
54     logits = my_batch_norm_layer(logits_before_bn) instantiate
```

배치 정규화 (3)

- 배치 정규화 이전과 배치 정규화 이후의 accuracy 변화

```
0 Train accuracy: 0.96 Validation accuracy: 0.902
1 Train accuracy: 0.9 Validation accuracy: 0.924
2 Train accuracy: 0.94 Validation accuracy: 0.9306
3 Train accuracy: 0.94 Validation accuracy: 0.9404
4 Train accuracy: 0.9 Validation accuracy: 0.9468
5 Train accuracy: 0.96 Validation accuracy: 0.9494
6 Train accuracy: 0.94 Validation accuracy: 0.9542
7 Train accuracy: 0.92 Validation accuracy: 0.957
8 Train accuracy: 0.9 Validation accuracy: 0.9594
9 Train accuracy: 1.0 Validation accuracy: 0.9616
```

```
0 validation accuracy: 0.9296
1 validation accuracy: 0.9548
2 validation accuracy: 0.9618
3 validation accuracy: 0.966
4 validation accuracy: 0.9674
5 validation accuracy: 0.971
6 validation accuracy: 0.972
7 validation accuracy: 0.9748
8 validation accuracy: 0.9752
9 validation accuracy: 0.9734
```

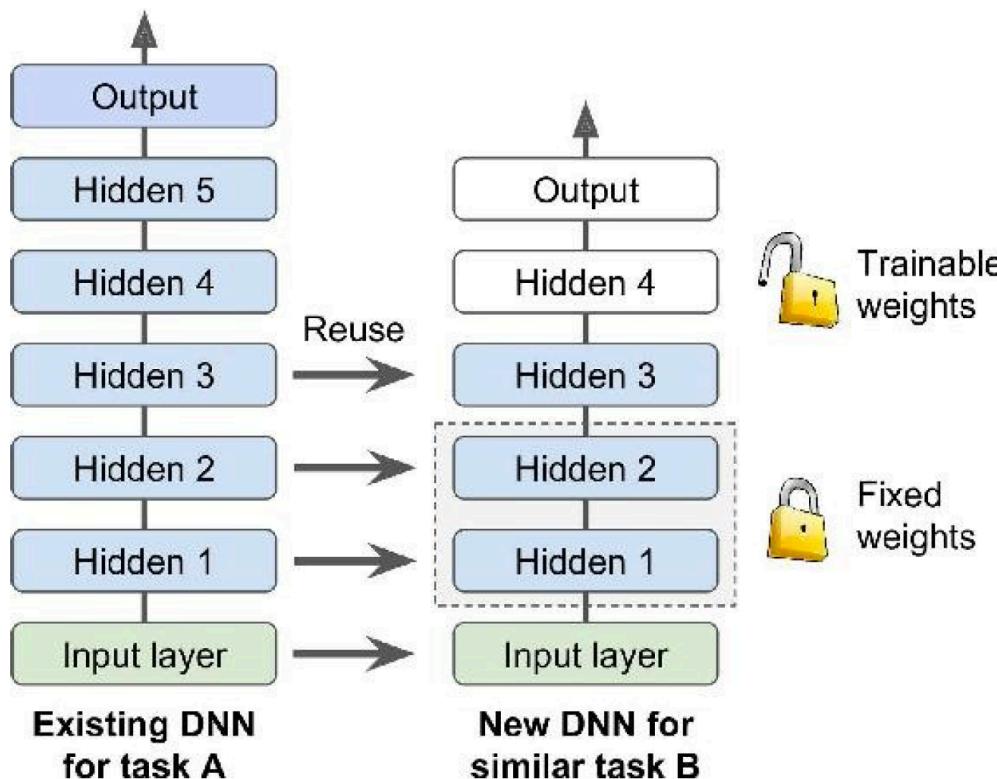
- Experiment Configuration

- 10장에서 설명한 심층 신경망 예제를 배치 정규화하여 비교
- 향후 실험 결과도 같은 실험 대상을 변경하여 비교
- 실험 환경 - MacBook

MacBook (Retina, 12-inch, Early 2015)
프로세서 1.3 GHz Intel Core M
메모리 8GB 1600 MHz DDR3

미리 훈련된 층 재사용하기

- 큰 규모의 DNN을 처음부터 새로 훈련시키는 것은 좋지 못한 방법
- 비슷한 유형의 문제를 처리한 신경망이 있는지 찾아보고 하위층을 재사용하는 것이 좋음



- ex) 다양한 사물을 분리하는 DNN의 하위층을 이용하여 자동차를 세분화하는 DNN을 생성
- 하위 3개층은 기존 모델의 것을 사용
- 새로운 상위층을 추가하여 학습
- 기존 모델의 하위층은 가중치를 고정

미리 훈련된 층 재사용하기 - in TensorFlow

- 기존 모델 저장 방법

```
78 saver = tf.train.Saver()  
109 save_path = saver.save(sess, "./my_model_final.ckpt");
```

- 저장된 모델 Load

```
7 # [LOAD existing model if exists]  
8 saver = tf.train.import_meta_graph("./my_model_final.ckpt.meta")
```

- 모델에 저장된 정보 참조

```
10 # 1. Train & target placeholder  
11 X = tf.get_default_graph().get_tensor_by_name("X:0");  
12 y = tf.get_default_graph().get_tensor_by_name("y:0");  
13 accuracy = tf.get_default_graph().get_tensor_by_name("eval/accuracy:0");  
14 training_op = tf.get_default_graph().get_tensor_by_name("GradientDescent");
```

- 모델에 저장된 모든 정보 참조

```
16 # can check every names  
17 for op in tf.get_default_graph().get_operations():  
18     print(op.name)
```

```
save/RestoreV2_16/shape_and_slices  
save/RestoreV2_16  
save/Assign_16  
save/RestoreV2_17/tensor_names  
save/RestoreV2_17/shape_and_slices  
save/RestoreV2_17  
save/Assign_17  
save/restore_all
```

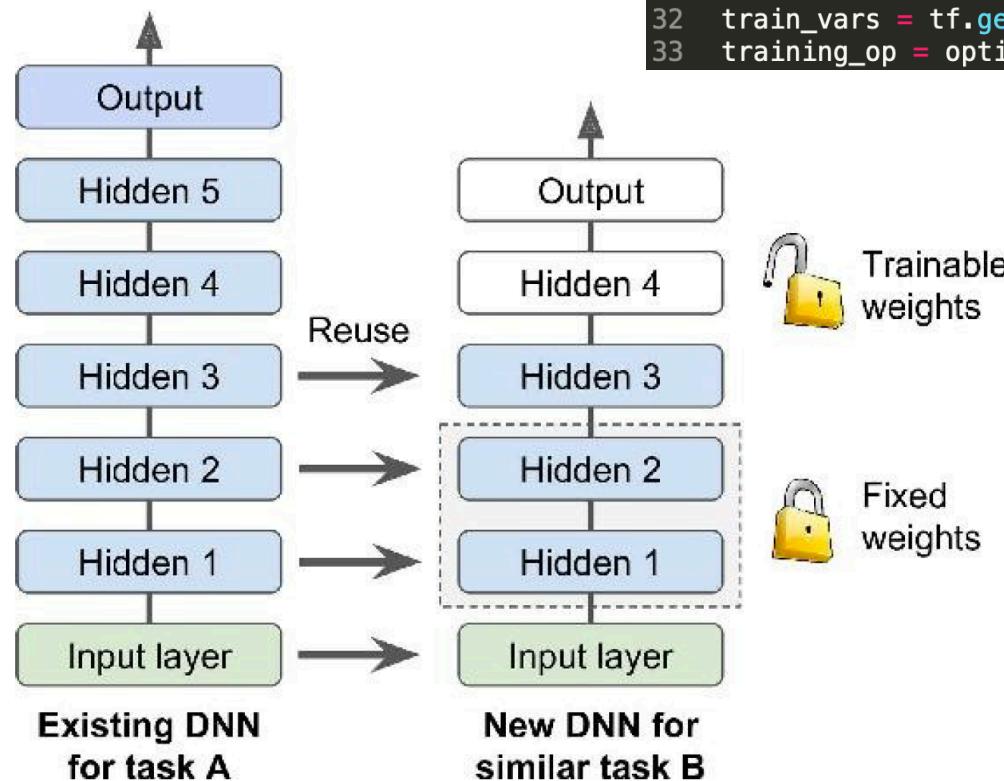
- Pre-defined Collection Save/Load

```
20 # How to save for others  
21 for op in (X, y, accuracy, training_op):  
22     tf.add_to_collection("my_important_ops", op)
```

```
24 # How to load for others  
25 X, y, accuracy, training_op, logits = tf.get_collection("my_important_ops")
```

미리 훈련된 층 재사용하기 - in TensorFlow

- 재사용 층들은 가중치를 동결하는 것이 좋음



```
32 train_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, scope="hidden[34] | outputs")  
33 training_op = optimizer.minimize(loss, var_list=train_vars)
```

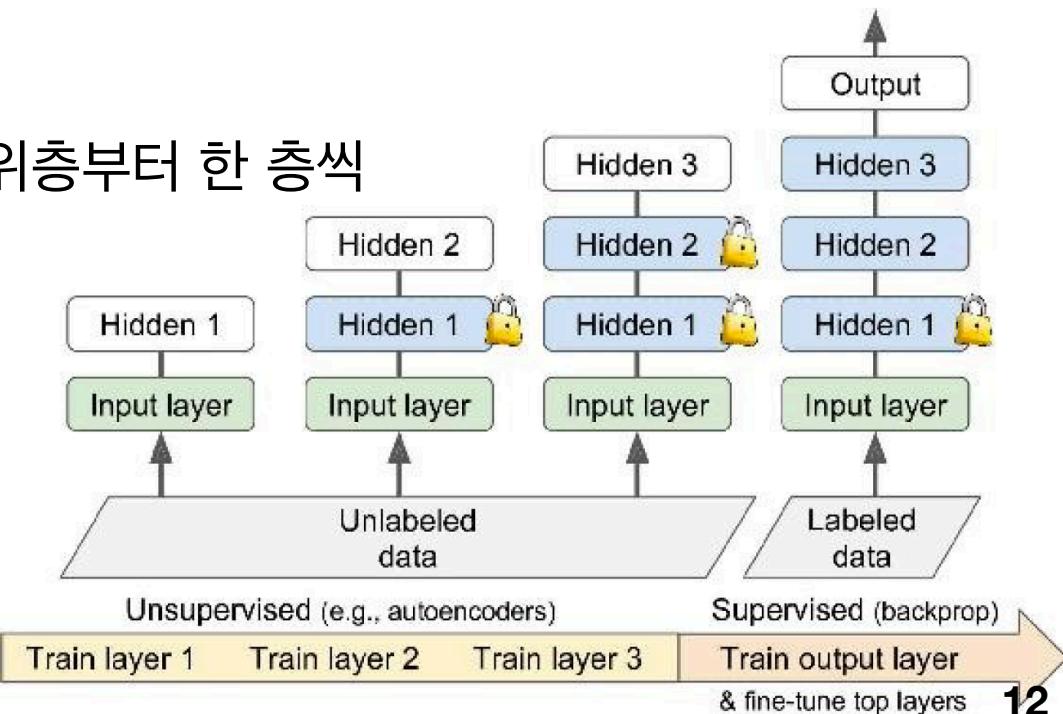
// train 과정에서 3, 4, 출력 층에 있는 변수들만을 대상으로 훈련

- 동결층 캐싱: 전체 훈련 세트에 대해 최상위 동결층의 결과를 캐싱하여 활용, 훈련 연산에 캐싱된 결과를 사용

```
import numpy as np  
  
n_batches = mnist.train.num_examples // batch_size  
  
with tf.Session() as sess:  
    init.run()  
    restore_saver.restore(sess, "./my_model_final.ckpt")  
  
    h2_cache = sess.run(hidden2, feed_dict={X: mnist.train.images})  
  
    for epoch in range(n_epochs):  
        shuffled_idx = np.random.permutation(mnist.train.num_examples)  
        hidden2_batches = np.array_split(h2_cache[shuffled_idx], n_batches)  
        y_batches = np.array_split(mnist.train.labels[shuffled_idx], n_batches)  
        for hidden2_batch, y_batch in zip(hidden2_batches, y_batches):  
            sess.run(training_op, feed_dict={hidden2: hidden2_batch, y: y_batch})  
  
    save_path = saver.save(sess, "./my_new_model_final.ckpt")
```

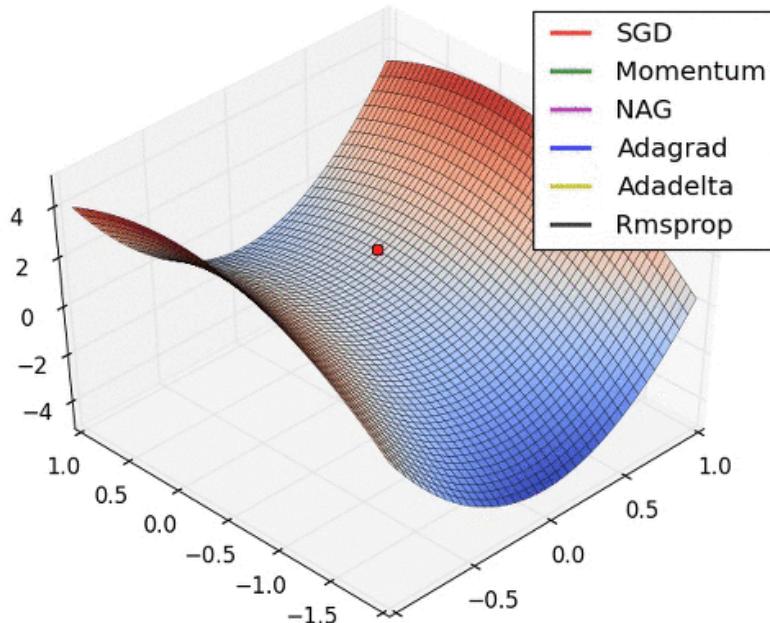
미리 훈련된 층 재사용하기 - other methods

- 모델 저장소를 활용하여 비슷한 작업을 훈련한 모델을 찾을 수 있다.
 - TensorFlow - <https://github.com/tensorflow/models>
 - 카페 모델 저장소 - <http://goo.gl/XI02X3>
- 비지도 사전 훈련을 사용
 - 레이블된 훈련 데이터가 많지 않은 경우 하위층부터 한 층씩 비지도 특성 추출 알고리즘을 사용하여 학습 시킬 수 있다.

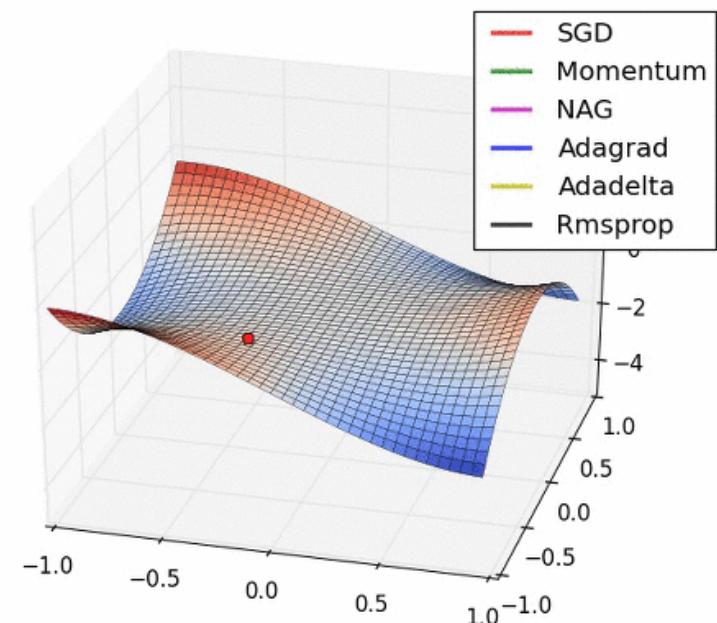


Faster Optimizers

- Optimizer를 사용하여 효과적인 학습을 수행
 - Momentum Optimization / NAG / Adagrad / RMSProp / Adam



<협곡에서의 Optimizer별 경로>



<말 안장점에서의 Optimizer별 경로>

Faster Optimizers

- 표준적인 경사하강법은 경사면을 따라 일정한 크기로 조금씩 내려가기 때문에 많은 학습시간을 요구
- 아래 두 방법은 기울기에 따라 다른 속도로 경사면을 따라간다.

Momentum Optimization

Equation 11-4. Momentum algorithm

$$\begin{aligned} 1. \quad \mathbf{m} &\leftarrow \beta \mathbf{m} - \eta \nabla_{\theta} J(\theta) \\ 2. \quad \theta &\leftarrow \theta + \mathbf{m} \end{aligned}$$

- 경사면에서 공이 미끄러지는 것과 같은 물리 법칙
 - J : 비용함수 / β : 하이퍼파라미터 / n : 학습률
- 다음 속도는 현재 위치의 기울기와 연관이 있음

NAG(Nesterov Momentum Optimization)

Equation 11-5. Nesterov Accelerated Gradient algorithm

$$\begin{aligned} 1. \quad \mathbf{m} &\leftarrow \beta \mathbf{m} - \eta \nabla_{\theta} J(\theta + \beta \mathbf{m}) \\ 2. \quad \theta &\leftarrow \theta + \mathbf{m} \end{aligned}$$

- Momentum Optimization의 확장
- 경사면을 따라서 조금 더 내려간 위치의 기울기를 사용

Faster Optimizers

Momentum Optimization

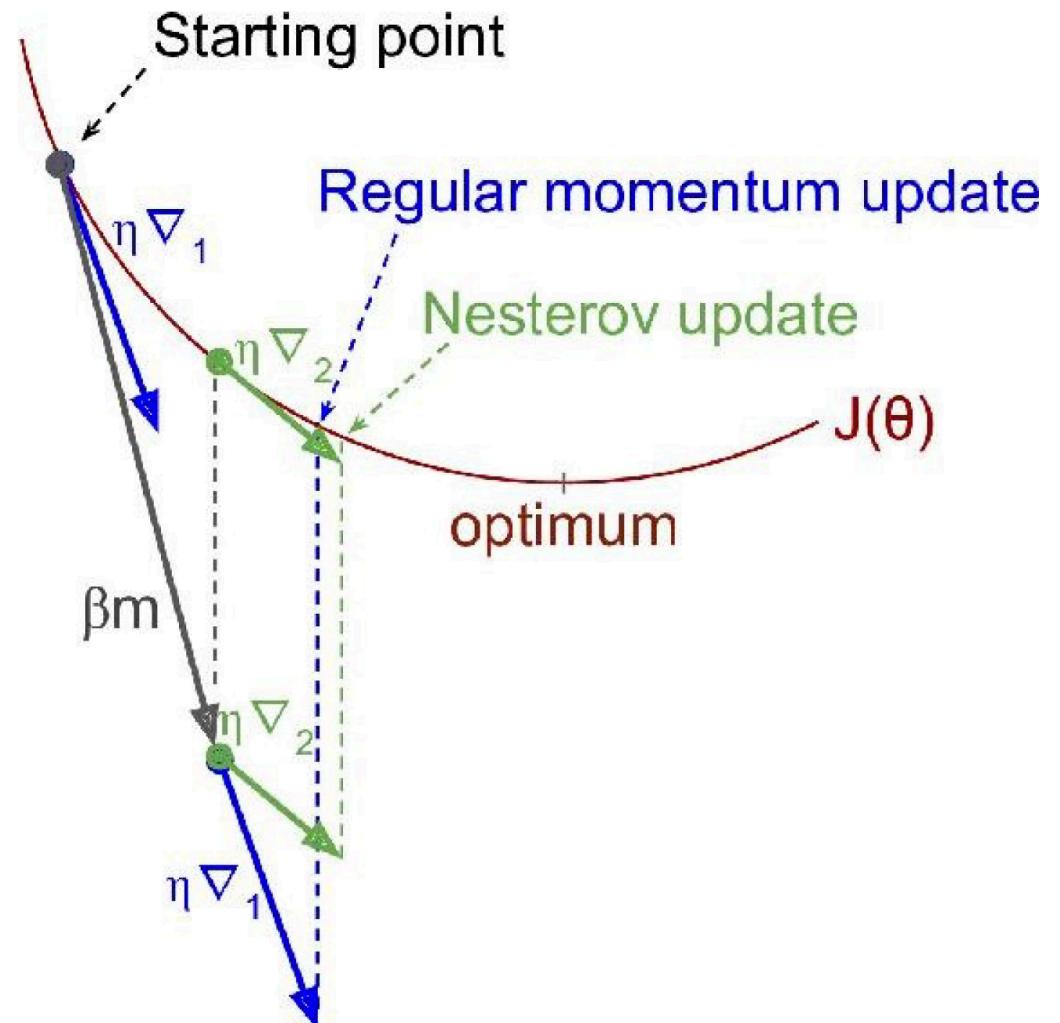
Equation 11-4. Momentum algorithm

1. $\mathbf{m} \leftarrow \beta\mathbf{m} - \eta \nabla_{\theta} J(\theta)$
2. $\theta \leftarrow \theta + \mathbf{m}$

NEG(Nesterov Momentum Optimization)

Equation 11-5. Nesterov Accelerated Gradient algorithm

1. $\mathbf{m} \leftarrow \beta\mathbf{m} - \eta \nabla_{\theta} J(\theta + \beta\mathbf{m})$
2. $\theta \leftarrow \theta + \mathbf{m}$



Faster Optimizers - Adaptive

- 적응적 학습률을 이용한 Optimizer
- 경사가 가장 가파른 차원을 따라 벡터의 스케일을 조정

AdaGrad

$$\begin{aligned}\mathbf{s} &\leftarrow \mathbf{s} + \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta) \\ \theta &\leftarrow \theta - \eta \nabla_{\theta} J(\theta) \oslash \sqrt{\mathbf{s} + \epsilon}\end{aligned}$$

- 훈련과정 중 학습률이 너무 일찍 감소되어 전역 최적점에 도착하기 전에 멈춤

RMSProp

$$\begin{aligned}\mathbf{s} &\leftarrow \beta \mathbf{s} + (1 - \beta) \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta) \\ \theta &\leftarrow \theta - \eta \nabla_{\theta} J(\theta) \oslash \sqrt{\mathbf{s} + \epsilon}\end{aligned}$$

- 가장 최근 반복된 그래디언트만 누적함으로써 AdaGrad의 문제를 해결

Adam Optimization

$$\begin{aligned}\mathbf{m} &\leftarrow \beta_1 \mathbf{m} - (1 - \beta_1) \nabla_{\theta} J(\theta) \\ \mathbf{x} &\leftarrow \beta_2 \mathbf{x} + (1 - \beta_2) \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta) \\ \mathbf{m} &\leftarrow \frac{\mathbf{m}}{1 - \beta_1^T} \\ \mathbf{x} &\leftarrow \frac{\mathbf{x}}{1 - \beta_2^T} \\ \theta &\leftarrow \theta + \eta \mathbf{m} \oslash \sqrt{\mathbf{x} + \epsilon}\end{aligned}$$

- Momentum 최적화와 RMSProp의 아이디어를 합쳐서 만든 알고리즘

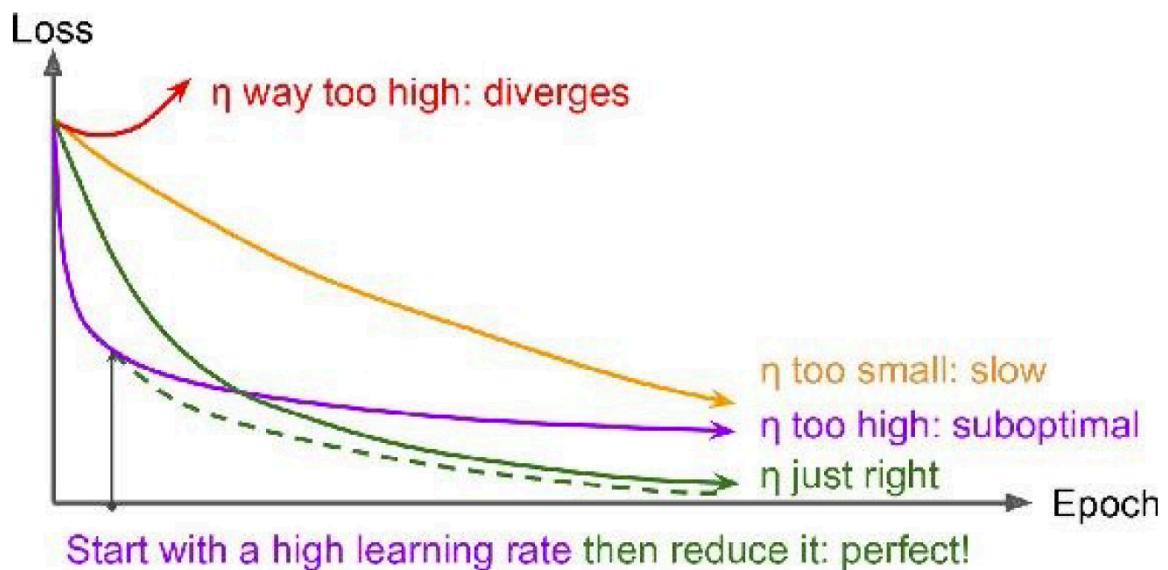
Faster Optimizers - Comparison

	NAG	Adam
Without Batch Normalization	<pre>0 validation accuracy: 0.9052 1 validation accuracy: 0.9234 2 validation accuracy: 0.9326 3 validation accuracy: 0.9416 4 validation accuracy: 0.9442 5 validation accuracy: 0.9526 6 validation accuracy: 0.957 7 validation accuracy: 0.9606 8 validation accuracy: 0.9628 9 validation accuracy: 0.9648</pre>	<pre>0 validation accuracy: 0.9538 1 validation accuracy: 0.9756 2 validation accuracy: 0.976 3 validation accuracy: 0.9748 4 validation accuracy: 0.9768 5 validation accuracy: 0.9776 6 validation accuracy: 0.9752 7 validation accuracy: 0.9824 8 validation accuracy: 0.9796 9 validation accuracy: 0.9826</pre>
With Batch Normalization	<pre>0 validation accuracy: 0.955 1 validation accuracy: 0.9746 2 validation accuracy: 0.9758 3 validation accuracy: 0.9786 4 validation accuracy: 0.9788 5 validation accuracy: 0.9796 6 validation accuracy: 0.9804 7 validation accuracy: 0.9794 8 validation accuracy: 0.9816 9 validation accuracy: 0.9826</pre>	<pre>0 validation accuracy: 0.9568 1 validation accuracy: 0.9748 2 validation accuracy: 0.9748 3 validation accuracy: 0.9798 4 validation accuracy: 0.981 5 validation accuracy: 0.9802 6 validation accuracy: 0.9814 7 validation accuracy: 0.982 8 validation accuracy: 0.9836 9 validation accuracy: 0.982</pre>

- Adam 방식이 더욱 빠르게 증가
- Batch Normalization을 적용한 경우에는 큰 차이가 없음
(복잡한 모델일 수록 차이가 클 것)

Faster Optimizer - 학습률 스케줄링

- 좋은 학습률을 찾는 것은 중요하지만 찾는 것이 쉽지 않다.
- 학습률이 너무 클 경우 발산할 수 있으며 너무 낮으면 느리게 학습이 될 수 있다.
- 일정한 학습률을 사용하는 것 보다 더 나은 방법은 처음에는 높은 학습률을 이용하여 학습한 후 낮은 학습률을 이용하여 학습하는 것



과대 적합을 피하기 위한 규제 방법

1. 조기종료 - 검증 세트의 성능이 떨어지기 시작할 때 종료
 1. 일정 step마다 모델을 평가하고 이전의 성능보다 나을 경우 저장
 2. 저장 이후 한계 step까지 좋은 모델이 생성되지 않으면 훈련 종료
2. 드롭아웃 - 매 훈련 step에서 각 뉴런은 p 확률(일반적으로 50%)로 활성화되지 않게 한다.
 - 최고 성능을 내는 신경망도 이를 이용하여 오차율을 추가적으로 40% 줄일 수 있다.
3. 데이터 증식 - 기존 데이터에서 새로운 데이터를 생성하여 인공적으로 훈련 데이터를 증식
 - ex) 사진의 피사체를 회전/이동함으로써 증식할 수 있음
4. 맥스-노름 규제 - 입력 가중치의 상한선을 둠

실용적 가이드라인

- 어떤 기법을 써야할지 고민된다면 대부분의 경우에 잘맞는 아래 설정을 사용할 수 있다.
- 다음의 경우에는 위의 기본 설정을 변경하여 사용
 - 훈련 세트가 적은 경우에는 데이터 증식을 적용
 - 회소 모델이 필요하다면 L1 규제를 추가
 - 실행 속도가 아주 빠른 모델이 필요하다면 배치 정규화를 빼고 ELU 대신 LeakyReLU로 교체
 - 좋은 학습률을 찾을 수 없는 경우 - 지수 감소와 같은 학습 스케줄 추가

초기화	He 초기화
활성화 함수	ELU
정규화	배치 정규화
규제	드롭아웃
옵티마이저	네스테로프 가속 경사 (NAG)