# *Optimizers*

Glenn Bruns
CSUMB

Much material in this deck from Géron, Hands-on Machine Learning with Scikit-Learn and TensorFlow

## Learning outcomes

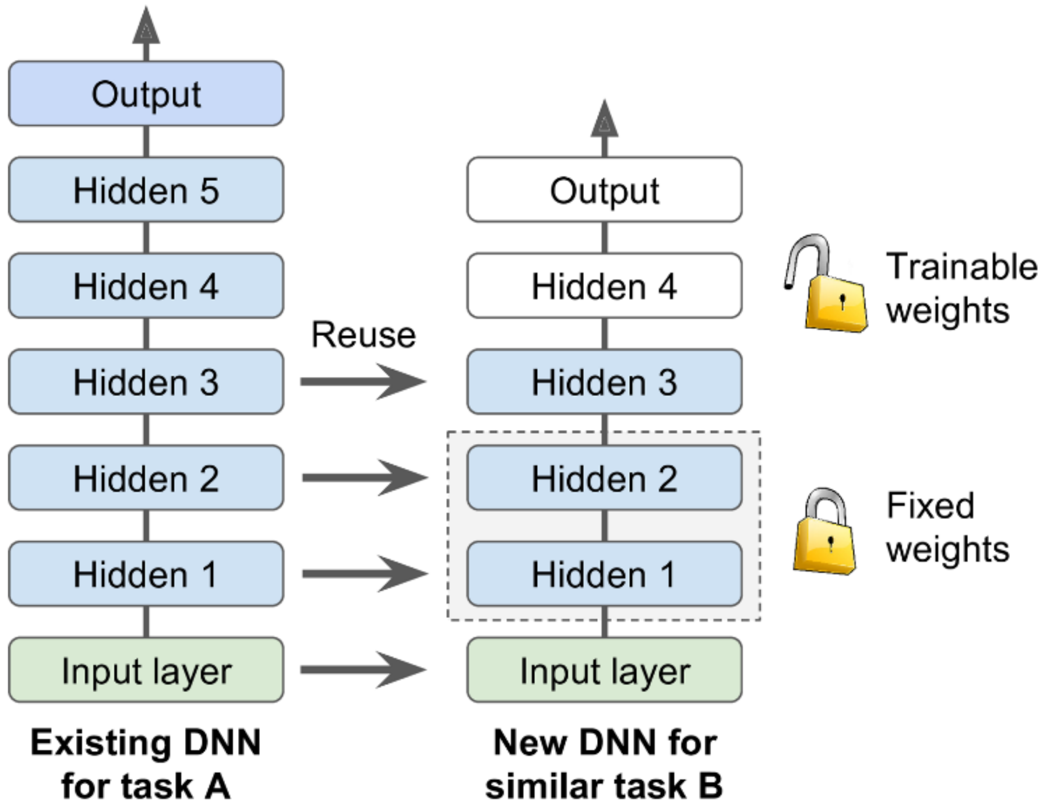
After this lecture you should be able to:

- reuse parts of TensorFlow models
- select a fast optimizer

# Reusing layers

- Training large nets takes a long time
- Many nets solve similar tasks

Transfer learning: reuse lower levels of a net solving a problem similar to your own



Example: you want to classify vehicles from images; existing nets classify images of animals, vehicles, etc.

## TensorFlow mechanics

### In TF, how to reuse only part of  model?

```
[...] # build the new model with the same hidden layers 1-3 as before

reuse_vars = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES,
                               scope="hidden[123]")
reuse_vars_dict = dict([(var.op.name, var) for var in reuse_vars])
restore_saver = tf.train.Saver(reuse_vars_dict) # restore layers 1-3

init = tf.global_variables_initializer() # init all variables
saver = tf.train.Saver() # to save the entire new model
```

```
with tf.Session() as sess:
    init.run()
    restore_saver.restore(sess, "./my_model_final.ckpt")
    [...] # train the model
    save_path = saver.save(sess, "./my_new_model_final.ckpt")
```

## Freezing layers

If you reuse weights from another DNN, you probably want to "freeze" them.

This means their values won't be involved in the training process.

An easy way to do this in TensorFlow: tell optimizer which variables to train:

```
train_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,
                               scope="hidden[34]|outputs")

# variables in hidden layers won't change during training
training_op = optimizer.minimize(loss, var_list=train_vars)
```

(see our text for an alternative method)

## Caching layers

If the reused layers are frozen, you only need to compute the output of the topmost frozen layer once.

Strategy:

- run the whole training set through the lower levels
- don't batch training data – batch outputs from the topmost frozen layer
- use these during training

## Strategies for tweaking reuse

- we know we don't want to reuse the output layer of original model
- how many of the hidden layers should be reused?
  - the higher the level, the less likely to be used
- strategy:
  - start by freezing all borrowed layers
  - train your model, see how it performs
  - if performance not very good, unfreeze one or two top borrowed layers
  - if performance still not very good, drop or replace the topmost borrowed layers

## Model zoos

- Where to find DNNs that solve tasks similar to your own?
  - your own previous DNNs (so keep them organized)
  - a model zoo – public repository of DNNs
- Model zoos:
  - TensorFlow's zoo: `github.com/tensorflow/models`
  - Caffe's zoo: `github.com/BVLC/caffe/wiki/Model-Zoo`
- The TF zoo contains most of the state-of-the-art image classification nets
- The "official models" in the TF zoo are supposed to be easy to read while still achieving high performance.

no picture of an animal here

## Unsupervised pretraining

- Suppose you have a difficult task and not much labeled training data
- Suppose also it's very hard to get more training data…
- … but you have some unlabeled training data
- Idea:
  - train levels one by one, starting at the lowest
  - use an unsupervised feature detector algorithm (e.g. Restricted Boltzmann Machines) or autoencoders
  - once all layers are trained, fine tune using supervised learning

Hinton's team revived deep learning in 2006 using this approach. Since 2010, problems with vanishing gradients were reduced and backprop again became commonly used.

## Pretraining on an auxiliary task

- Similar to the idea of borrowing layers from another DNN
- Idea 1:
  - you want to build a system to classify faces, but have little labeled training data
  - so grab a bunch of unlabeled face images, and training then to distinguish whether two pictures are of the same person
  - then reuse lower levels of this DNN for your problem
- Idea 2:
  - you are building a language processing DNN
  - grab a bunch of English sentences, label them 'good'
  - corrupt a bunch of English sentences, label them 'bad'
  - train a classifier to predict which sentences are good or bad

## Faster optimizers

- We have looked at speedups from:
  - better initialization of weights
  - better activation functions
  - batch normalization
  - reusing parts of a pretrained network
- Now we'll look at optimizers that are faster than standard gradient descent optimization:
  - Momentum optimization, Nesterov Accelerated Gradient, AdaGrad, RMSProp, Adam optimization
  - See text for info on AdaGrad, RMSProp

## Gradient descent

Question: how important do you think gradient descent is to this course?

1. not so important
2. kind of important
3. very important

# Gradient descent

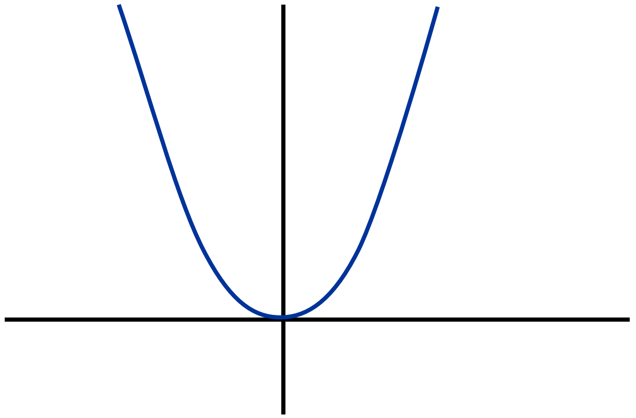
Question: write down the update rule for gradient descent, given a single parameter $\theta$, a cost function $J(\theta)$. We are looking for value of $\theta$ that minimizes the cost function.

Hint: you can write the learning rate as variable $\eta$.

$$\theta = \theta - \eta \frac{dJ}{d\theta}\,\theta$$

Question: if $J(\theta) = \theta^2$ , $\theta = 1$, and $\eta = 0.2$, then what is the updated value of $\theta$ ?

Answer: $\frac{dJ}{d\theta} = 2\theta$, so
$\theta = 1 - (0.1)(2) = 0.8$



# Issues with gradient descent

Recall the standard gradient descent update rule:

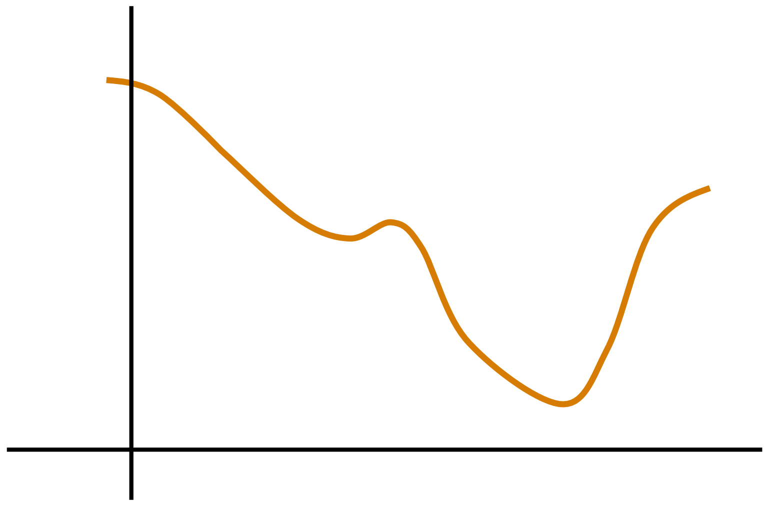
$$\theta = \theta - \eta \, \nabla_\theta J(\theta)$$

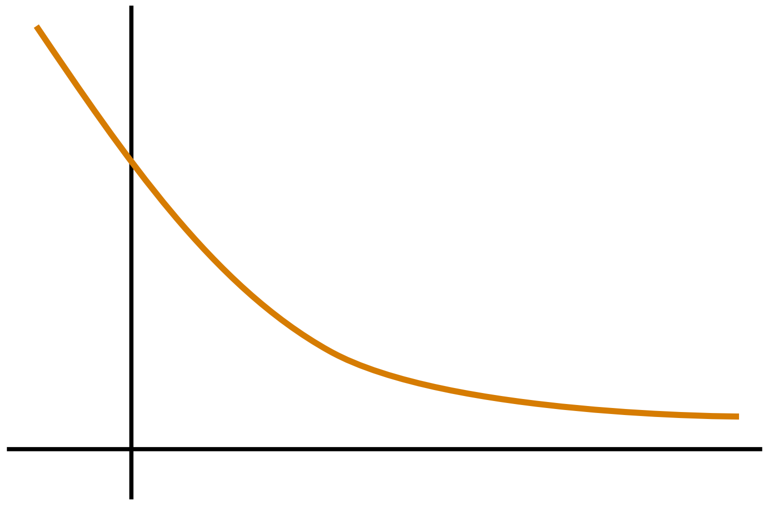
where $J(\theta)$ is the cost function.

The change to $\theta$ depends only on the local gradient.

Example:

$$\eta = 0.1, \ \ \nabla_\theta J(\theta) = (1{,}1)$$
$$\theta = (0.5, 2)$$
$$\theta = (0.4, 1.9)$$
$$\theta = (0.3, 1.8)$$





# Momentum optimization

Idea: gain momentum on downhill slopes and "roll past" local minima

$$\mathbf{m} = \beta\mathbf{m} + \eta\,\nabla_\theta\, J(\theta)$$
$$\theta = \theta - \mathbf{m}$$

where $\mathbf{m}$ is the momentum vector.

$0 \le \beta \le 1$ 0 means "high friction", 1 means "no friction"

Suppose gradient is constant, $\beta$ = 0.9 (typical value)

Example:
$\eta = 0.1$
$\nabla_\theta J(\theta) = (1,1)$
$\mathbf{m} = (0,0)$

$\theta = (0.5, 2)$
$\mathbf{m} = 0 + (0.1, 0.1)$
$\theta = (0.4, 1.9)$
$\mathbf{m} = ?$
$\theta = ?$

$\theta = (0.5, 2)$
$\mathbf{m} = 0 + (0.1, 0.1)$
$\theta = (0.4, 1.9)$
$\mathbf{m} = (0.09 + 0.09) + (0.1, 0.1)$
$\quad = (0.19, 0.19)$
$\theta = (0.21, 1.71)$

## Nesterov Accelerated Gradient

- A variant of Momentum optimization
- Idea: measure gradient at position slightly ahead of direction of momentum

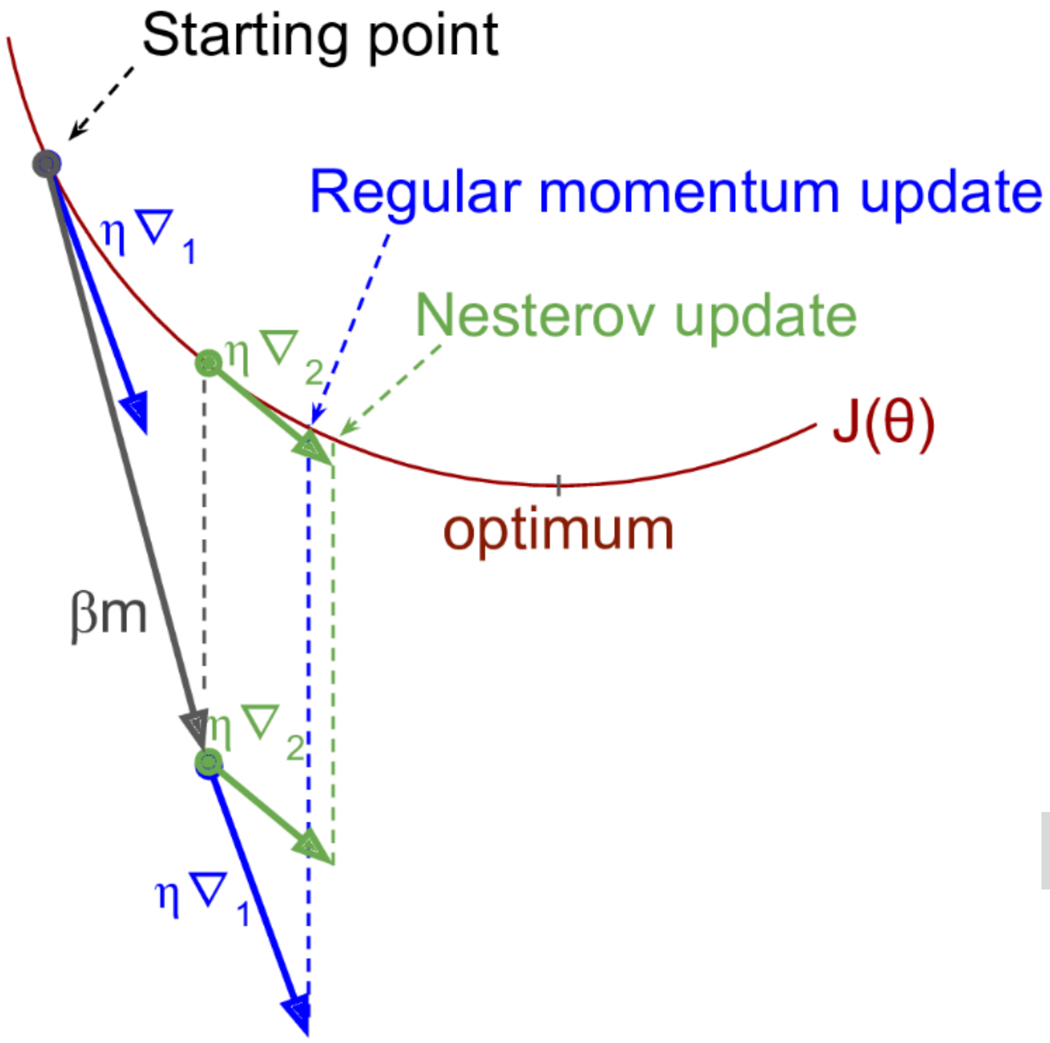
$$\mathbf{m} = \beta\mathbf{m} + \eta\, \nabla_\theta\, J(\theta + \beta\mathbf{m})$$
$$\theta = \theta - \mathbf{m}$$

- In plain Momentum optimization we had

$$\mathbf{m} = \beta\mathbf{m} + \eta\, \nabla_\theta\, J(\theta)$$

"look where you're going"

# Momentum vs. Nesterov



momentum:

$$\mathbf{m} = \beta\mathbf{m} + \eta\, \nabla_\theta\, J(\theta)$$
$$\theta = \theta - \mathbf{m}$$

Nesterov:

$$\mathbf{m} = \beta\mathbf{m} + \eta\, \nabla_\theta\, J(\theta + \beta\mathbf{m})$$
$$\theta = \theta - \mathbf{m}$$

## AdaGrad, RMSProp, Adam Optimization

- AdaGrad
  - adjusts learning rate so that learning rate is reduced in dimensions of greatest descent
  - "adaptive learning rate"
  - helps to avoid getting stuck in a ravine
- RMSProp
  - helps fix problem that AdaGrad often stops too early
  - usually outperforms Momentum, NAG, and AdaGrad
- Adam Optimization
  - combines Momentum and RMSProp
  - adaptive learning rate
  - works great; robust with respect to hyperparameters

## Tensorflow

- momentum

```
optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate,
momentum=0.9)
```

- Nesterov accelerated gradient

```
optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate,
momentum=0.9, use_nesterov=True)
```

- Adam Optimization

```
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
```

## Summary

Tricks to speed training:

- reusing layers, freezing layers, caching layers
- strategies for tweaking reuse
- model zoos
- unsupervised pretraining
- pretraining on an auxiliary task

Improved optimization algorithms

- Momentum, Nesterov Accelerated Gradient
- AdaGrad, RMSProp
- Adam Optimization