intel Nervana™ AI Academy

ALEXNET AND OPTIMIZERS

ALEXNET

ALEXNET

Created in 2012 for the ImageNet Large Scale Visual Recognition Challenge (ILSVRC)

Task: predict the correct label from among 1000 classes

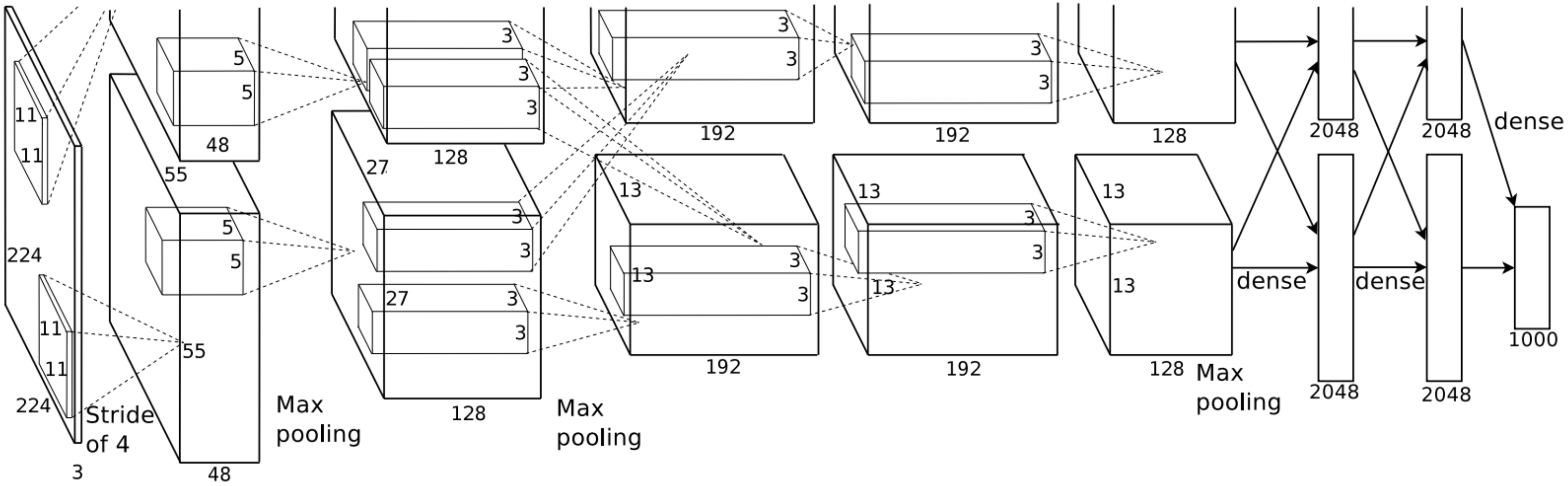
Dataset: around 1.2 million images

Considered the "flash point" for modern deep learning

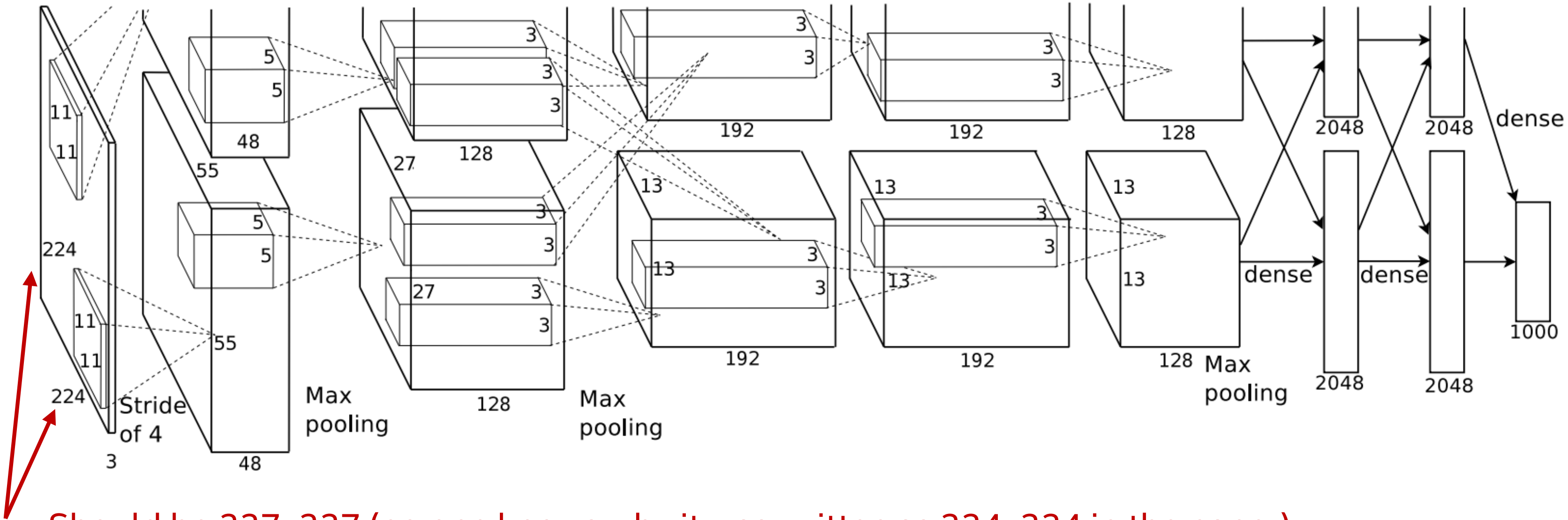
Demolished the competition.

- Top 5 error rate of 15.4%
- Next best: 26.2%

MODEL DIAGRAM



MODEL DIAGRAM



Should be 227x227 (no one knows why it was written as 224x224 in the paper)

NOTES

They perform data augmentation for training

- Cropping, horizontal flipping, and more
- Useful to help make more use out of given training data

They split up the model across two GPUs, as illustrated in previous image

- This generally doesn't happen in modern CNN architectures
- We can replicate this effect by splitting Tensors in two

ALEXNET: MAIN TAKEAWAYS

CNNs are *very* powerful for image processing

Didn't change too much about LeNet-5

- Added extra depth, computation

Basic template:

- Convolutions with ReLUs
- Sometimes add maxpool after convolutional layer
- Fully connected layers at the end before a softmax classifier

GPUs are really good for this sort of computation!

SAVING AND LOADING MODELS

SAVING TENSORFLOW MODELS

So far: our TensorFlow models have been transient

- We build, train, and play with them. Then they poof into the ether

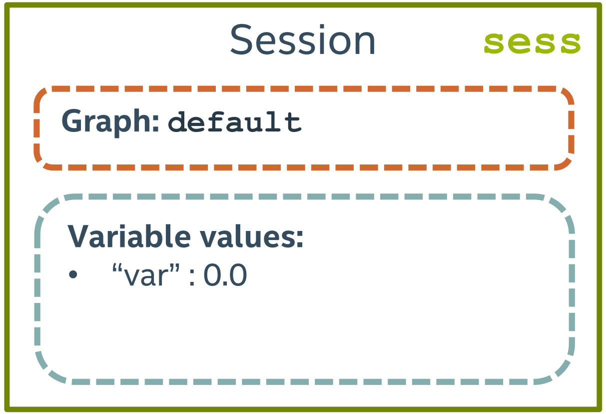
We need to be able to save our models for later use!

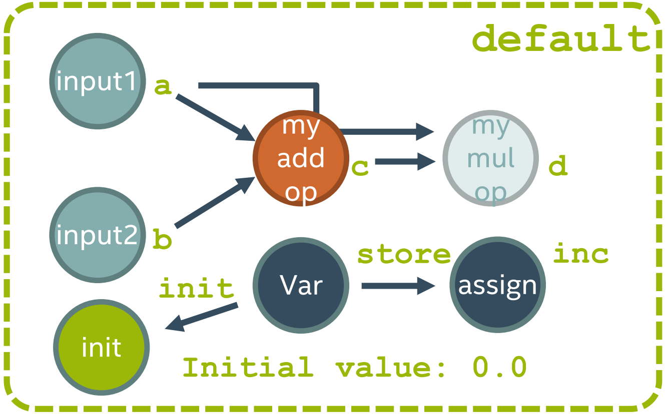
TensorFlow has built in mechanisms for saving/restoring

HOW TENSORFLOW STORES DATA

Recall that TensorFlow keeps the Graph definition separate from the current values of Variables

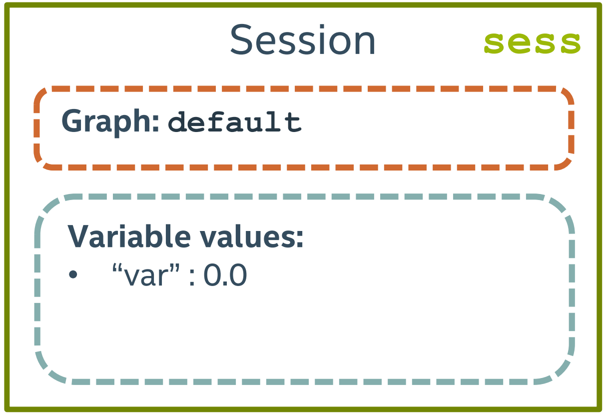
The same thing occurs with saving data to disk.

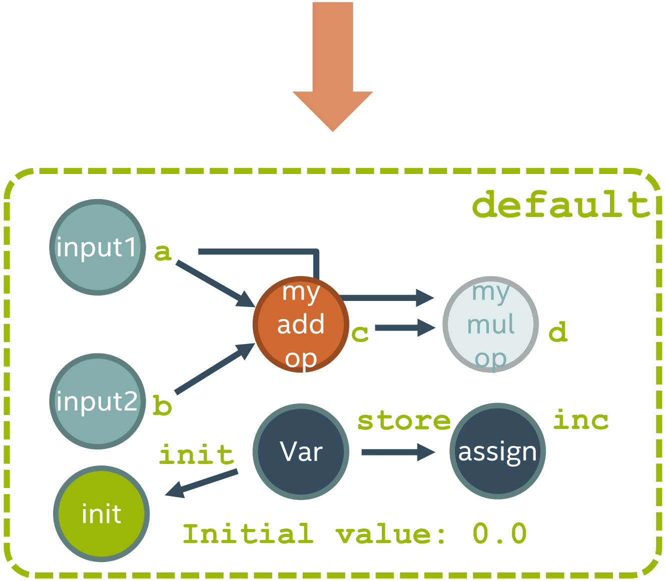




HOW TENSORFLOW STORES DATA

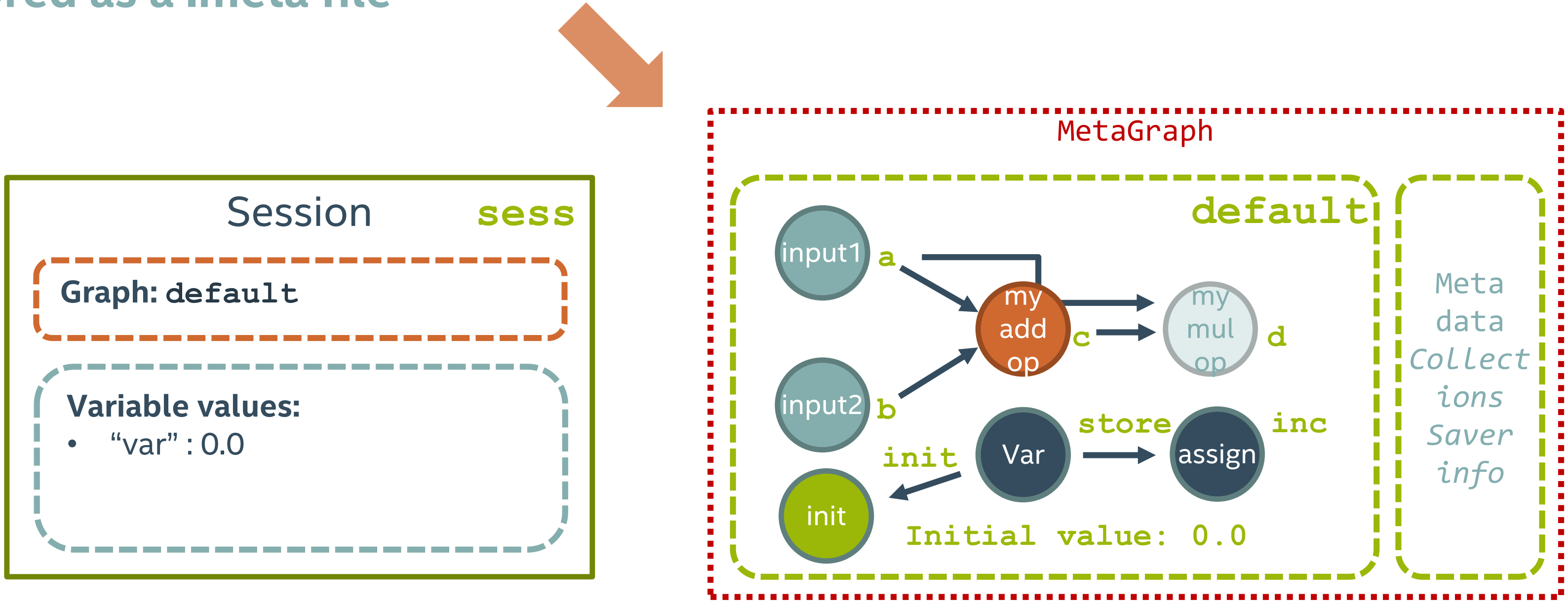
Graph info (ops, connections, etc) is stored in a GraphDef protocol buffer





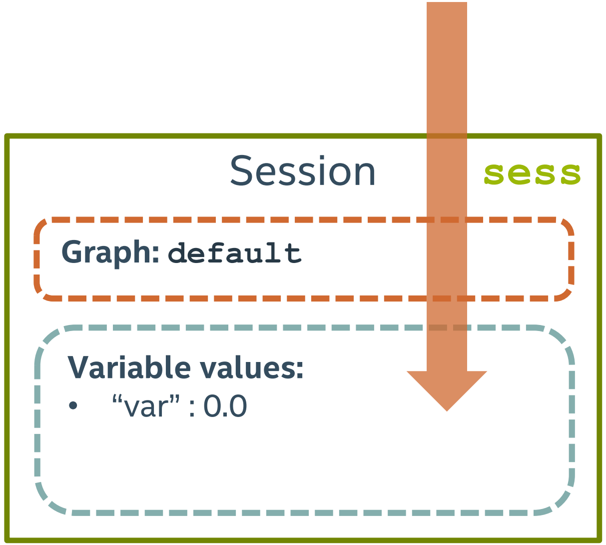
HOW TENSORFLOW STORES DATA

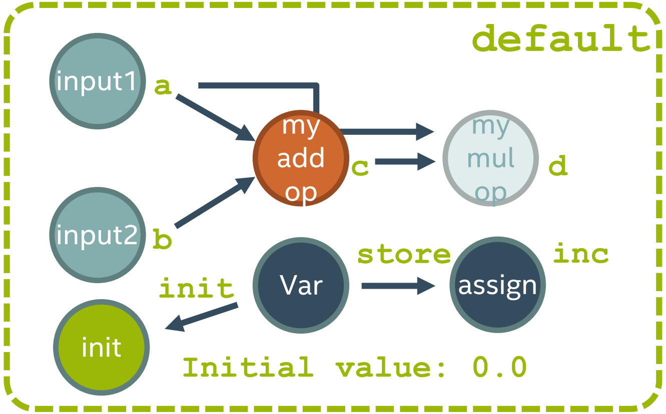
A MetaGraph encapsulates the graph definition along with relevant meta data. Stored as a .meta file



HOW TENSORFLOW STORES DATA

Variable state (weights, biases, etc) is stored in two files: a .index file and a .data file (older file format = .ckpt)





THE SAVER CLASS

The Saver class is designed to manage saving and loading both Variable checkpoints and MetaGraphs

The simplest use case when saving:

```
with graph.as_default():
    ..create a graph, define some variables
saver = tf.train.Saver()
with tf.Session(graph=graph) as sess:
    ..train the model
    saver.save(sess, './my_model')
```

THE SAVER CLASS

Then, to load a model:

```
new_graph = tf.Graph()
with new_graph.as_default():
    saver = tf.train.import_meta_graph('./my_model.meta')
with tf.Session(graph=new_graph) as sess:
    saver.restore(sess, './my_model')
    ..continue training
```

SAVING MULTIPLE CHECKPOINTS OVER TIME

You can pass in a global_step to the Saver.save() method

- Adds a numeric suffix to the exported files, e.g. ‘my_model-100’
- Allows you to easily save versions of a trained model over time

```
saver.save(sess, './my_model', global_step=global_step)
```

You can automatically get the latest version name with `tf.train.latest_checkpoint()`

```
saver.restore(sess, tf.train.latest_checkpoint('./'))
```

OPTIMIZER ALTERNATIVES

STANDARD UPDATE RULE FOR GRADIENT DESCENT

Recall our weight update with gradient descent

$$W := W - \alpha \cdot \Delta W$$

Can we change this update rule to speed up training?

IDEA 1: MOMENTUM

Assuming our error curve is bowl-ish shaped, can assume we'll be going in roughly the same direction over time

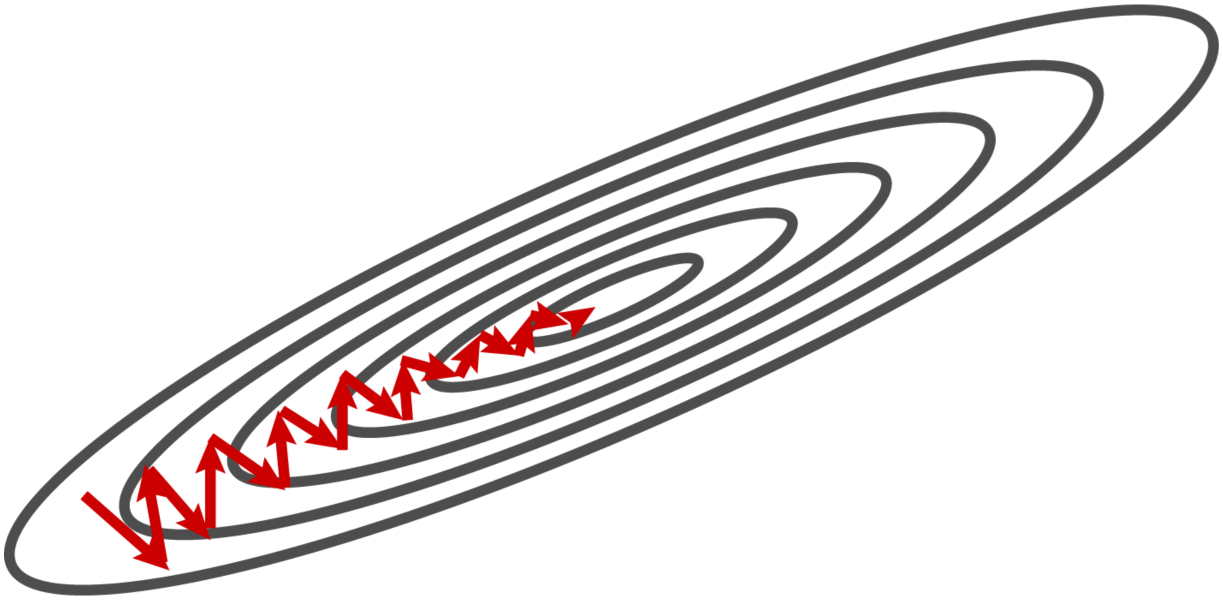
We alter our weight update by a factor of previous update

$$v_{\mathrm{t}} := \eta \cdot v_{t-1} - \alpha \cdot \Delta W$$

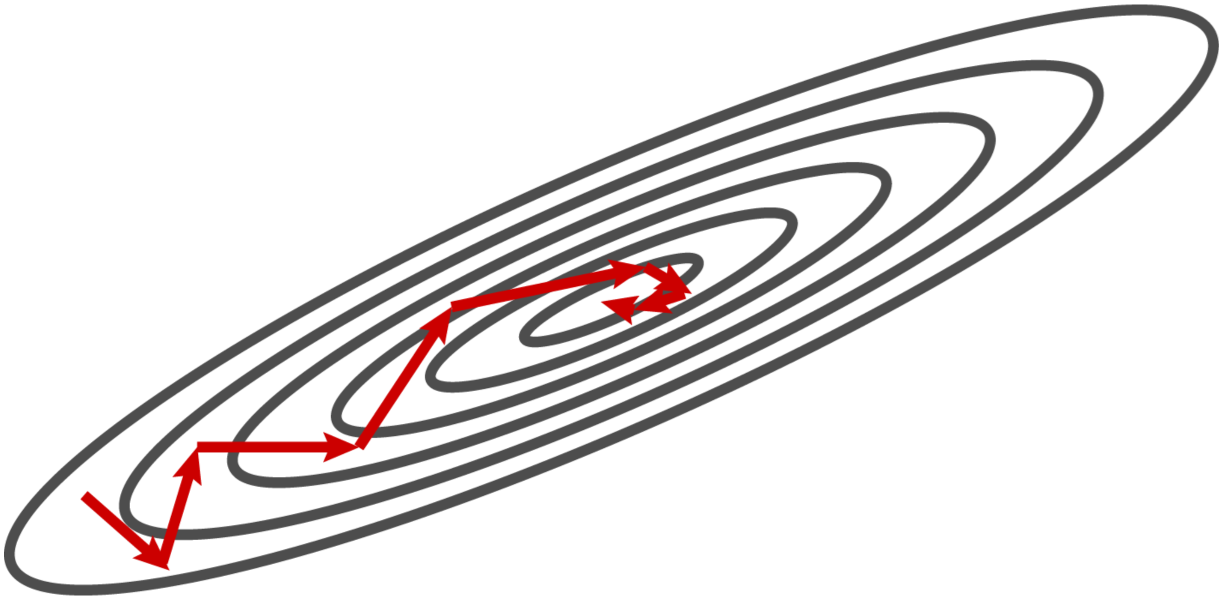
$$\mathrm{W} := W - v_t$$

η is often referred to as the "momentum"

WITHOUT MOMENTUM



WITH MOMENTUM



NESTEROV MOMENTUM

Momentum might accidentally "roll up the other side of the hill"

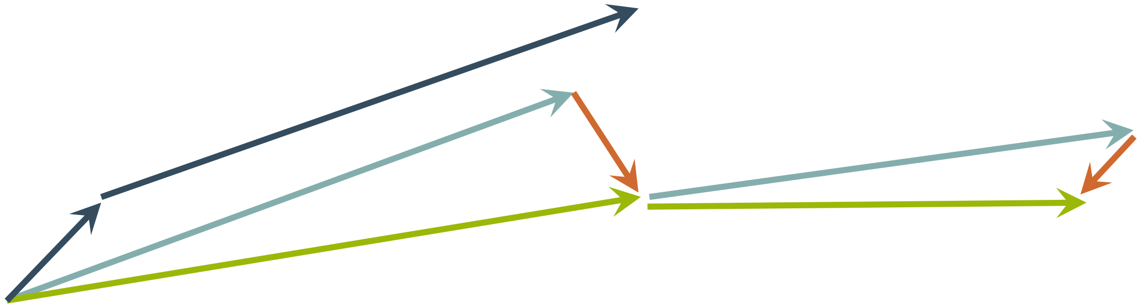
Nesterov momentum looks ahead before updating weights

$$u_t = \eta \cdot v_{t-1}$$

$$v_t = u_t - \alpha \cdot \Delta(W - u_t)$$

$$\mathrm{W} \coloneqq W - v_t$$

NESTEROV MOMENTUM



Momentum Vector

Nesterov steps

Gradient/correction

Standard momentum steps

[Source: Lecture by Geoffrey Hinton]

ADAGRAD

Idea: scale each weight's updates separately

Update frequently-updated weights less

Keep running tally of previous updates

Divide new updates by factor of previous tally

$$W := W - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot \Delta W$$

- G_t - Accumulated sum of squares for each individual ΔW
- Downside: eventually, all weights diminish to zero

ADADELTA AND RMSPROP

Variation on AdaGrad- seeks to reduce diminishing gradients

- Developed separately, but very similar algorithms

Basic idea: decay squared gradients (instead of full sum)

RMSProp update:

$$G_t = \gamma \cdot G_{t-1} + (1-\gamma)\Delta W^2$$

$$W := W - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot \Delta W$$

Note: In AdaDelta, γ (gamma/momentum) is ρ (rho) as named parameter in TensorFlow

ADAM

Idea: decaying tally of both sum squares and regular sum of weight updates:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\Delta W \qquad v_t = \beta_2 v_{t-1} + (1 - \beta_2)\Delta W^2$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \qquad \hat{v}_t = \frac{v_t}{1 - \beta_1^t}$$

$$W \coloneqq W - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \odot \hat{m}_t$$

GOOD NEWS!

TensorFlow has optimizers built in:

```
tf.train.MomentumOptimizer()
tf.train.MomentumOptimizer(…, use_nesterov=True)
tf.train.AdagradOptimizer()
tf.train.AdadeltaOptimizer()
tf.train.AdamOptimizer()
```

[Link to API documentation]

WHICH TO USE?

Many papers use vanilla momentum, with η around 0.9

RMSProp is supposedly good for RNNs

Adam is generally a very strong choice overall

Great blog post on optimizers by Sebastian Ruder

- Really great visualizations of learning

Nervana™ AI Academy