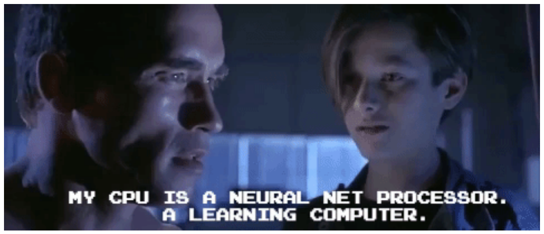
CS388: Natural Language Processing

Lecture 4: Neural Networks

Greg Durrett

TEXAS
The University of Texas at Austin



Administrivia

- Project 1 due Tuesday, final pieces for Part 2 covered today
- Project 2 due date pushed back to Feb 13
- FP check-in due date listed on course website (April 4)

Recall: Multiclass Classification

- Two views of multiclass classification:
  - Different features: $\text{argmax}_{y \in \mathcal{Y}} w^\top f(x, y)$
  - Different weights: $\text{argmax}_{y \in \mathcal{Y}} w_y^\top f(x)$
- Logistic regression: $P_{\mathbf{w}}(y = \hat{y} \mid \mathbf{x}) = \frac{\exp\left(\mathbf{w}_{\hat{y}}^\top \mathbf{f}(\mathbf{x})\right)}{\sum_{y'} \exp\left(\mathbf{w}_{y'}^\top \mathbf{f}(\mathbf{x})\right)}$

Gradient of log likelihood:

“increase value for gold weight vector, decrease for other weight vectors”

$$\frac{\partial}{\partial \mathbf{w}_{y^{(i)}}} \mathcal{L}(\mathbf{x}^{(i)}, y^{(i)}) = \mathbf{f}(\mathbf{x}^{(i)})(P_{\mathbf{w}}(y^{(i)} \mid \mathbf{x}^{(i)}) - 1)$$

$$\frac{\partial}{\partial \mathbf{w}_{\tilde{y}}} \mathcal{L}(\mathbf{x}^{(i)}, y^{(i)}) = \mathbf{f}(\mathbf{x}^{(i)}) P_{\mathbf{w}}(y^{(i)} \mid \mathbf{x}^{(i)})$$

This Lecture

- Neural network basics
- Feedforward neural networks + backpropagation
- Deep averaging network (Project 1)
- Implementing neural networks, training tips

Neural Net Basics

Neural Networks

- Linear classification: $\text{argmax}_y w^\top f(x,y)$
- Want to learn intermediate conjunctive features of the input

*the movie was **not** all that **good***

I[contains *not* & contains *good*]

- How do we learn this if our feature vector is just the unigram indicators?

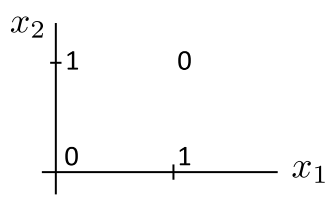
I[contains *not*], I[contains *good*]

Neural Networks: XOR

- Let's see how we can use neural nets to learn a simple nonlinear function
- Inputs $x_1, \; x_2$

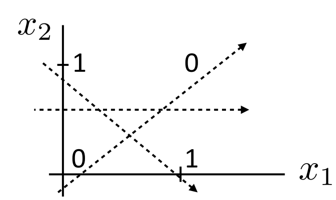
  (generally $\mathbf{x} = (x_1, \ldots, x_m)$)
- Output y

  (generally $\mathbf{y} = (y_1, \ldots, y_n)$)



x_1	x_2	$y = x_1 \text{ XOR } x_2$
0	0	0
0	1	1
1	0	1
1	1	0

Neural Networks: XOR

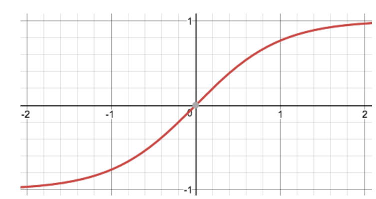


$y = a_1x_1 + a_2x_2$ X

$y = a_1x_1 + a_2x_2 + a_3\tanh(x_1 + x_2)$ ✓

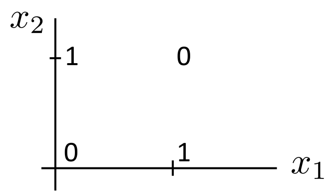
"or"

(looks like action potential in neuron)



x_1	x_2	$x_1 \text{ XOR } x_2$
0	0	0
0	1	1
1	0	1
1	1	0

Neural Networks: XOR



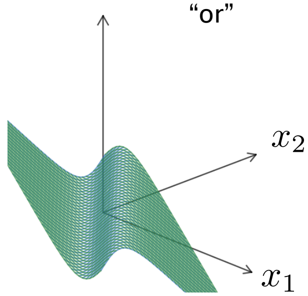
x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0

$y = a_1 x_1 + a_2 x_2$ X

$y = a_1 x_1 + a_2 x_2 + a_3 \tanh(x_1 + x_2)$ ✓

$y = -x_1 - x_2 + 2\tanh(x_1 + x_2)$

"or"



Neural Networks

Linear model: $y = \mathbf{w} \cdot \mathbf{x} + b$

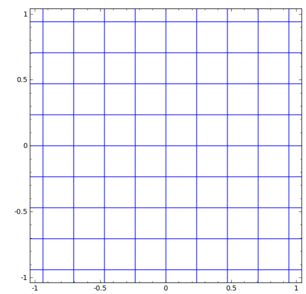
$$y = g(\mathbf{w} \cdot \mathbf{x} + b)$$

$$\mathbf{y} = g(\mathbf{W}\mathbf{x} + \mathbf{b})$$

Nonlinear transformation

Warp space

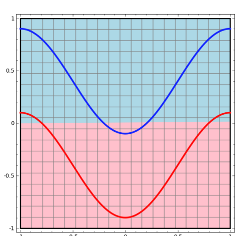
Shift



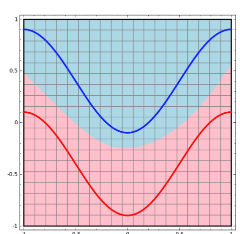
Taken from http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

Neural Networks

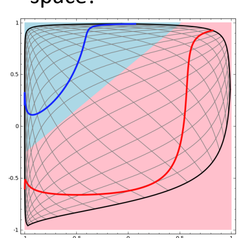
Linear classifier



Neural network



...possible because we transformed the space!



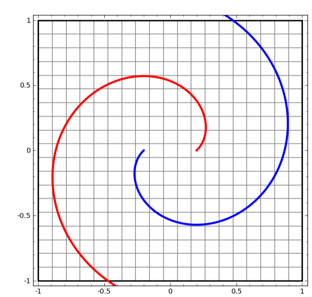
Taken from http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

Deep Neural Networks

$$\boldsymbol{y} = g(\mathbf{W}\boldsymbol{x} + \boldsymbol{b})$$

$$\mathbf{z} = g(\mathbf{V}\mathbf{y} + \mathbf{c})$$

$$\mathbf{z} = g(\mathbf{V}\underbrace{g(\mathbf{W}\mathbf{x} + \mathbf{b})}_{\text{output of first layer}} + \mathbf{c})$$



Check: what happens if no nonlinearity?

More powerful than basic linear models?

$$\mathbf{z} = \mathbf{V}(\mathbf{W}\mathbf{x} + \mathbf{b}) + \mathbf{c}$$

Taken from http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

Feedforward Networks, Backpropagation

Logistic Regression with NNs

$$P_{\mathbf{w}}(y = \hat{y} \mid \mathbf{x}) = \frac{\exp\left(\mathbf{w}_{\hat{y}}^\top \mathbf{f}(\mathbf{x})\right)}{\sum_{y'} \exp\left(\mathbf{w}_{y'}^\top \mathbf{f}(\mathbf{x})\right)}$$

- Single scalar probability

$$P(\mathbf{y} \mid \mathbf{x}) = \text{softmax}([\mathbf{w}_{\hat{y}}^\top \mathbf{f}(\mathbf{x})]_{y \in \mathcal{Y}})$$

- Compute scores for all possible labels at once (returns vector)

$$\text{softmax}(p)_i = \frac{\exp(p_i)}{\sum_{i'} \exp(p_{i'})}$$

- softmax: exps and normalizes a given vector

$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}(W f(\mathbf{x}))$$

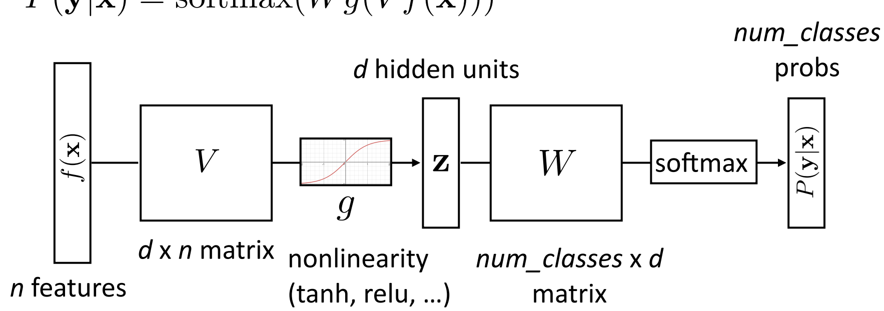
- Weight vector per class;
  W is [num classes x num feats]

$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}(W g(V f(\mathbf{x})))$$

- Now one hidden layer

Neural Networks for Classification

$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}(W g(V f(\mathbf{x})))$$



Training Neural Networks

$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}(W\mathbf{z}) \qquad \mathbf{z} = g(V f(\mathbf{x}))$$

- Maximize log likelihood of training data

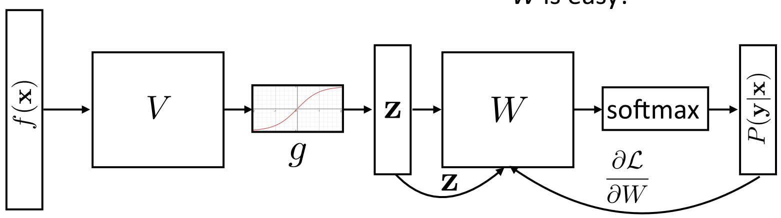
$$\mathcal{L}(\mathbf{x}, i^*) = \log P(y = i^*|\mathbf{x}) = \log\left(\text{softmax}(W\mathbf{z}) \cdot e_{i^*}\right)$$

- *i**: index of the gold label
- e_i: 1 in the *i*th row, zero elsewhere. Dot by this = select *i*th index
- This is exactly the same as logistic regression with ***z*** as the features!

Neural Networks for Classification

$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}(Wg(Vf(\mathbf{x})))$$

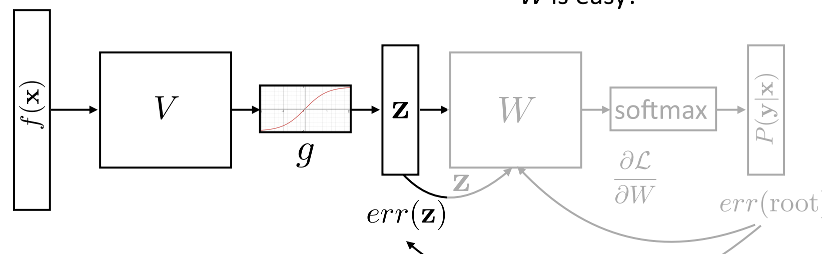
- Two sets of params to update: *W* and *V*. *W* is easy!



Backpropagation: Picture

$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}(Wg(Vf(\mathbf{x})))$$

- Two sets of params to update: *W* and *V*. *W* is easy!



- Can forget everything after **z**, treat it as the output and keep backpropping

Backpropagation: Takeaways

- Gradients of output weights *W* are easy to compute — looks like logistic regression with hidden layer ***z*** as feature vector
- Can compute derivative of loss with respect to ***z*** to form an "error signal" for backpropagation
- Easy to update parameters based on "error signal" from next layer, keep pushing error signal back as backpropagation
- Need to remember the values from the forward computation

Implementing NNs

(see ffnn_example.py on the course website)

Computation Graphs

- Computing gradients is hard! Computation graph abstraction allows us to define a computation symbolically and will do this for us
- Automatic differentiation: keep track of derivatives / be able to backpropagate through each function:

```
y = x * x   ──codegen──▶   (y,dy) = (x * x, 2 * x * dx)
```

- Use a library like Pytorch or Tensorflow. This class: Pytorch

Computation Graphs in Pytorch

- Define forward pass for $P(\mathbf{y}|\mathbf{x}) = \mathrm{softmax}(Wg(Vf(\mathbf{x})))$

```
class FFNN(nn.Module):
    def __init__(self, inp, hid, out):
        super(FFNN, self).__init__()
        self.V = nn.Linear(inp, hid)
        self.g = nn.Tanh()
        self.W = nn.Linear(hid, out)
        self.softmax = nn.Softmax(dim=0)

    def forward(self, x):
        return self.softmax(self.W(self.g(self.V(x))))
```

Computation Graphs in Pytorch

$$P(\mathbf{y}|\mathbf{x}) = \mathrm{softmax}(Wg(Vf(\mathbf{x})))$$

ei*: one-hot vector of the label (e.g., [0, 1, 0])

```
ffnn = FFNN()
def make_update(input, gold_label):
  ffnn.zero_grad() # clear gradient variables
  probs = ffnn.forward(input)
  loss = torch.neg(torch.log(probs)).dot(gold_label)
  loss.backward()
  optimizer.step()
```

Training a Model

Define a computation graph

For each epoch:

  For each batch of data:

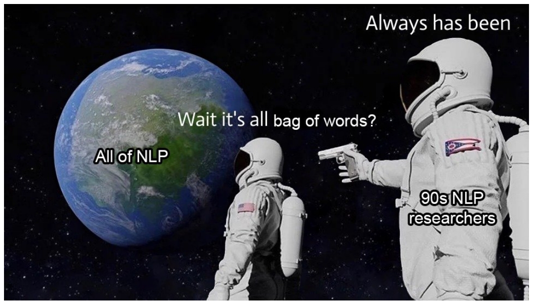
    Compute loss on batch

    Autograd to compute gradients

    Take step with optimizer

Decode test set

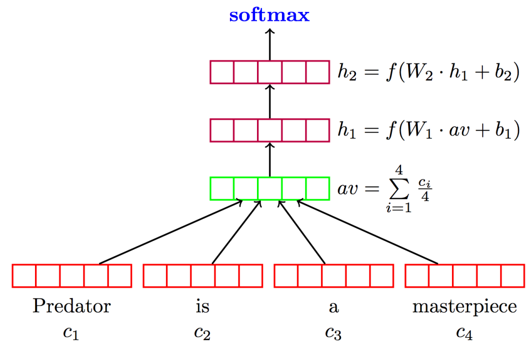
Deep Averaging Networks, Sentiment Analysis



Credit: Stephen Roller

Sentiment Analysis (Project 1)

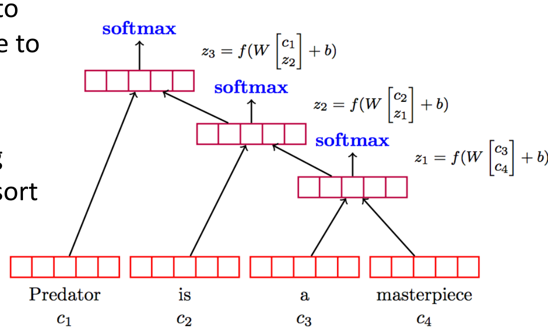
- Deep Averaging Networks: feedforward neural network on average of *word embeddings* from input



Iyyer et al. (2015)

Deep Averaging Networks

- Widely-held view: need to model syntactic structure to represent language
- Surprising that averaging can work as well as this sort of composition



Iyyer et al. (2015)

Sentiment Analysis

	Model	RT	SST fine	SST bin	IMDB	Time (s)	
	DAN-ROOT	—	46.9	85.7	—	**31**	
	DAN-RAND	77.3	45.4	83.2	88.8	136	
	DAN	80.3	47.7	86.3	89.4	136	Iyyer et al. (2015)
Bag-of-words	NBOW-RAND	76.2	42.3	81.4	88.9	91	
	NBOW	79.0	43.6	83.6	89.0	91	
	BiNB	—	41.9	83.1	—	—	Wang and Manning (2012)
	NBSVM-bi	79.4	—	—	91.2	—	
Tree RNNs / CNNS / LSTMS	RecNN*	77.7	43.2	82.4	—	—	
	RecNTN*	—	45.7	85.4	—	—	
	DRecNN	—	49.8	86.6	—	431	
	TreeLSTM	—	**50.6**	86.9	—	—	
	DCNN*	—	48.5	86.9	89.4	—	
	PVEC*	—	48.7	87.8	**92.6**	—	
	CNN-MC	**81.1**	47.4	**88.1**	—	2,452	Kim (2014)
	WRRBM*	—	—	—	89.2	—	

Word Embeddings in PyTorch

- torch.nn.Embedding: maps vector of indices to matrix of word vectors

Predator is a masterpiece
1820 24 1 2047
↓

- *n* indices => *n* x *d* matrix of *d*-dimensional word embeddings
- *b* x *n* indices => *b* x *n* x *d* tensor of *d*-dimensional word embeddings
- Steps to Project 1: define a module that takes a list of indices, then does the embedding + averaging and feeds the result through an FFNN (can use the module from ffnn_example.py as a starter)

Tips for Project 1

- Word embedding layer can be either frozen or trained — be attentive to this (torch.nn.Embedding layer from the WordEmbeddings class)
- As with the linear model, most minor tweaks like dropout, etc. will make <1% difference. If you're 10% off the performance target, it's likely due to a mis-sized network, poor optimization, bugs, etc.
- Debugging: follow ffnn_example.py, can use 50-dim embeddings to debug (they're smaller and a bit faster to use)

POS Tagging with FFNNs

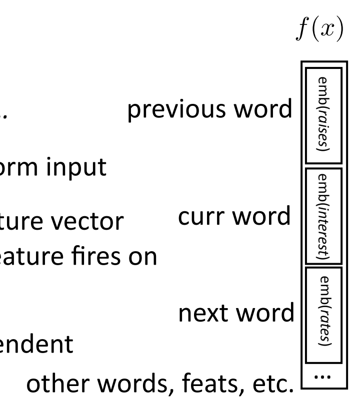
NLP with Feedforward Networks

- Part-of-speech tagging with FFNNs

??

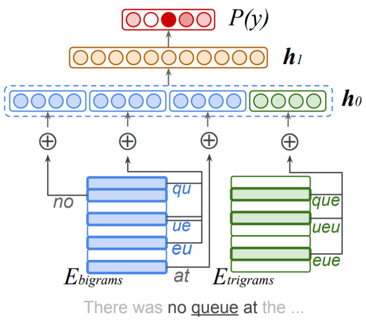
*Fed raises **interest** rates in order to …*

- Word embeddings for each word form input
- ~1000 features here — smaller feature vector than in sparse models, but every feature fires on every example
- Weight matrix learns position-dependent processing of the words



Botha et al. (2017)

NLP with Feedforward Networks



- Hidden layer mixes these different signals and learns feature conjunctions

Botha et al. (2017)

NLP with Feedforward Networks

- Multilingual tagging results:

Model	Acc.	Wts.	MB	Ops.
Gillick et al. (2016)	95.06	900k	-	6.63m
Small FF	94.76	241k	0.6	0.27m
+Clusters	95.56	261k	1.0	0.31m
$\frac{1}{2}$ Dim.	95.39	143k	0.7	0.18m

- Gillick used LSTMs; this is smaller, faster, and better

Botha et al. (2017)

Training Tips

Batching

- Batching data gives speedups due to more efficient matrix operations
- Need to make the computation graph process a batch at the same time

```
# input is [batch_size, num_feats]
# gold_label is [batch_size, num_classes]
def make_update(input, gold_label)
    ...
    probs = ffnn.forward(input) # [batch_size, num_classes]
    loss = torch.sum(torch.neg(torch.log(probs)).dot(gold_label))
    ...
```

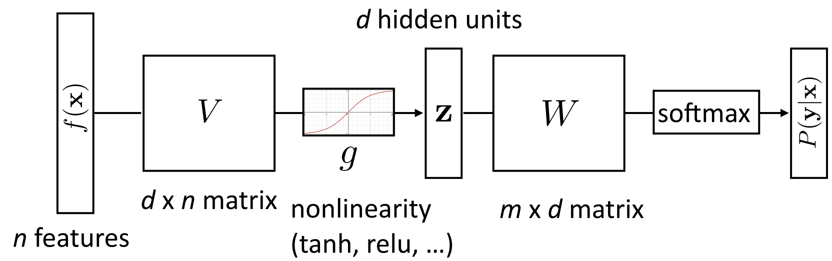
- Batch sizes from 1-100 often work well

Training Basics

- Basic formula: compute gradients on batch, use first-order optimization method (SGD, Adagrad, etc.)
- How to initialize? How to regularize? What optimizer to use?
- This lecture: some practical tricks. Take deep learning or optimization courses to understand this further

How does initialization affect learning?

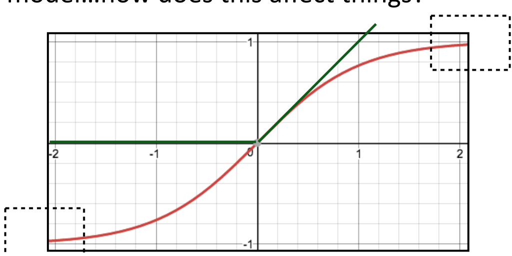
$$P(\mathbf{y}|\mathbf{x}) = \mathrm{softmax}(W g(V f(\mathbf{x})))$$



- How do we initialize V and W? What consequences does this have?
- *Nonconvex* problem, so initialization matters!

How does initialization affect learning?

- Nonlinear model…how does this affect things?



- If cell activations are too large in absolute value, gradients are small
- ReLU: larger dynamic range (all positive numbers), but can produce big values, can break down if everything is too negative

Initialization

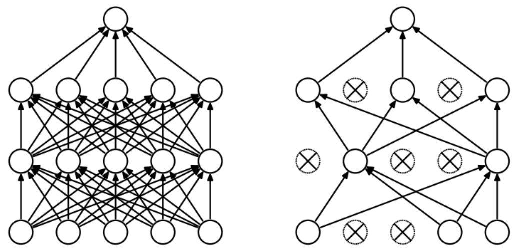
1) Can't use zeroes for parameters to produce hidden layers: all values in that hidden layer are always 0 and have gradients of 0, never change

2) Initialize too large and cells are saturated

- Can do random uniform / normal initialization with appropriate scale
- Glorot initializer: $U\left[-\sqrt{\frac{6}{\text{fan-in}+\text{fan-out}}}, +\sqrt{\frac{6}{\text{fan-in}+\text{fan-out}}}\right]$
  - Want variance of inputs and gradients for each layer to be the same
- Batch normalization (Ioffe and Szegedy, 2015): periodically shift+rescale each layer to have mean 0 and variance 1 over a batch (useful if net is deep)

Dropout

- Probabilistically zero out parts of the network during training to prevent overfitting, use whole network at test time
- Form of stochastic regularization
- Similar to benefits of ensembling: network needs to be robust to missing signals, so it has redundancy
- One line in Pytorch/Tensorflow



(a) Standard Neural Net (b) After applying dropout.

Srivastava et al. (2014)

Adam

$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)

- *m*: exponentially-weighted moving average of gradients
- *v*: exponentially-weighted moving average of gradients squared
- $\beta_1 = 0.9$, $\beta_2 = 0.999$, so these average over many steps
- Update is based on normalized corrected mean, incorporates *momentum*

Kingma and Ba (2015)

Next Time: Word Representations

Word Embeddings

- Currently we think of words as "one-hot" vectors

  the = [1, 0, 0, 0, 0, 0, …]

  good = [0, 0, 0, 1, 0, 0, …]

  great = [0, 0, 0, 0, 0, 1, …]

- *good* and *great* seem as dissimilar as *good* and *the*
- Neural networks are built to learn sophisticated nonlinear functions of continuous inputs; our inputs are weird and discrete

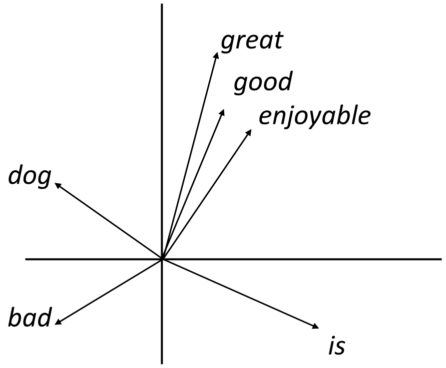
Word Embeddings

- Want a vector space where similar words have similar embeddings

the movie was great

≈

the movie was good



- Goal: come up with a way to produce these embeddings
- For each word, want "medium" dimensional vector (50-300 dims) representing it

Takeaways

- Feedforward neural networks can be implemented easily in PyTorch
  - We saw that these are basically logistic regression
  - Easy to implement backpropagation (you don't have to do anything!) and use the standard tricks to get good performance
- Next class: thinking about the feature representations: word representations / word vectors (word2vec and GloVe)