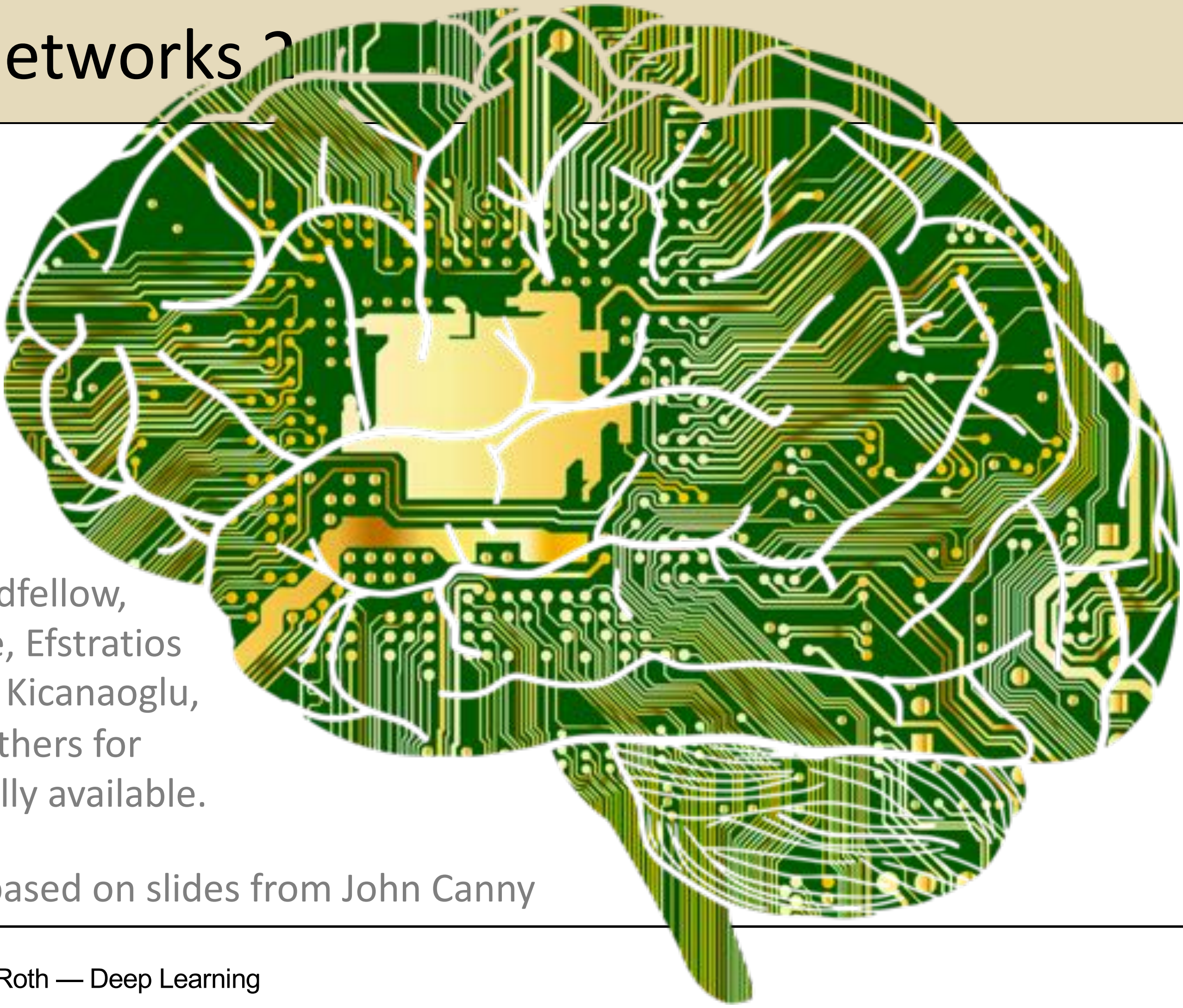
Deep Learning

Architectures and Methods

Training Neural Networks 2



Thanks to John Canny, Ian Goodfellow, Yoshua Bengio, Aaron Courville, Efstratios Gavves, Kirill Gavrilyuk, Berkay Kicanaoglu, and Patrick Putzky and many others for making their materials publically available.

The present slides are mainly based on slides from John Canny

Now: wrap up of training deep networks

Last time on training:

- Activation Functions
- Data Preprocessing
- Weight Initialization
- Batch Normalization
- Babysitting the Learning process

Now: wrap up of training deep networks

- Hyperparameter optimization
- Ensembles
- Dropout
- One-bit gradients
- Gradient noise

Ensemble Learning

Ensemble: A model built from many simpler models.

Two main methods:

- Bagging:

- Boosting:

Ensemble Learning

Ensemble: A model built from many simpler models

Two main methods:

- Bagging: (Bootstrap AGgregation): Train base models on bootstrap samples of the data. Take majority vote for classification tasks, or average output for regression.
- Boosting:

Ensemble Learning

Ensemble: A model built from many simpler models
Two main methods:

- Bagging: (Bootstrap AGgregation): Train base models on bootstrap samples of the data. Take majority vote for classification tasks, or average output for regression.

- Boosting: Learners are ordered: Each learner tries to reduce error (residual) on “hard” examples (those misclassified by earlier learners).

Ensemble Learning

Ensemble: A model built from many simpler models

Two main methods:

- Bagging: (Bootstrap AGgregation): Train base models on bootstrap samples of the data. Take majority vote for classification tasks, or average output for regression. Reduces variance in the prediction, not bias. Bootstrap sampling not always used – but some method is needed to generate diverse base models – e.g. random forests.

Ensemble Learning

Ensemble: A model built from many simpler models

Two main methods:

- Bagging: (Bootstrap AGgregation): Train base models on bootstrap samples of the data. Take majority vote for classification tasks, or average output for regression.

- Boosting: Learners are ordered: Each learner tries to reduce error (residual) on “hard” examples (those misclassified by earlier learners). ADABOOST: weight hard samples more; GRADIENT BOOST: use residual to train later models. Reduces bias and possibly variance compared to base learners.

Ensemble Learning

Ensemble: A model built from many simpler models

Two main methods:

- Bagging: (Bootstrap AGgregation): Train base models on bootstrap samples of the data. Take majority vote for classification tasks, or average output for regression.

- Boosting: Learners are ordered: Each learner tries to reduce error (residual) on “hard” examples (those misclassified by earlier learners).

- In both cases, the ensemble prediction is an evenly-weighted sum (or vote) of the base learner predictions.
- Aside: Stacking uses non-uniform base learner weighting.

Bagging on image classification tasks

Enjoy 2% extra accuracy

Ensemble Learning

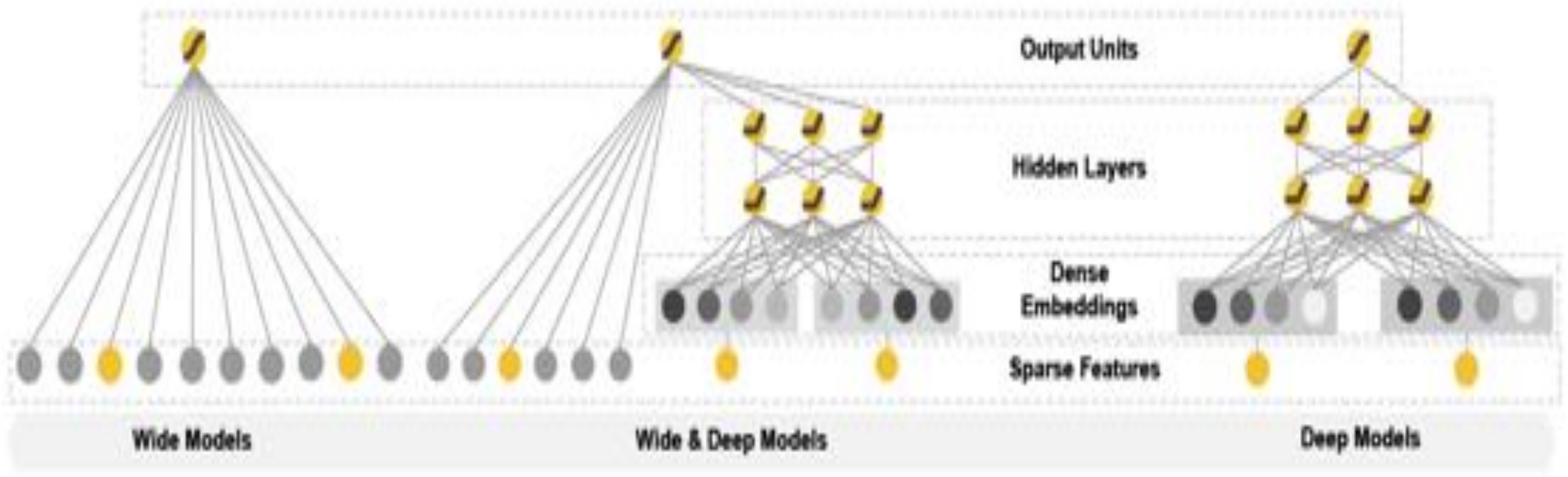
Gradient-boosted decision trees (GBDT) often gives state-of-the-art performance on simple classification tasks, e.g. XGBOOST.

Neural networks are used fairly often with bagging, but rarely with boosting. Why do you think that is?

Hint: Decision trees work well in both bagging and boosting.

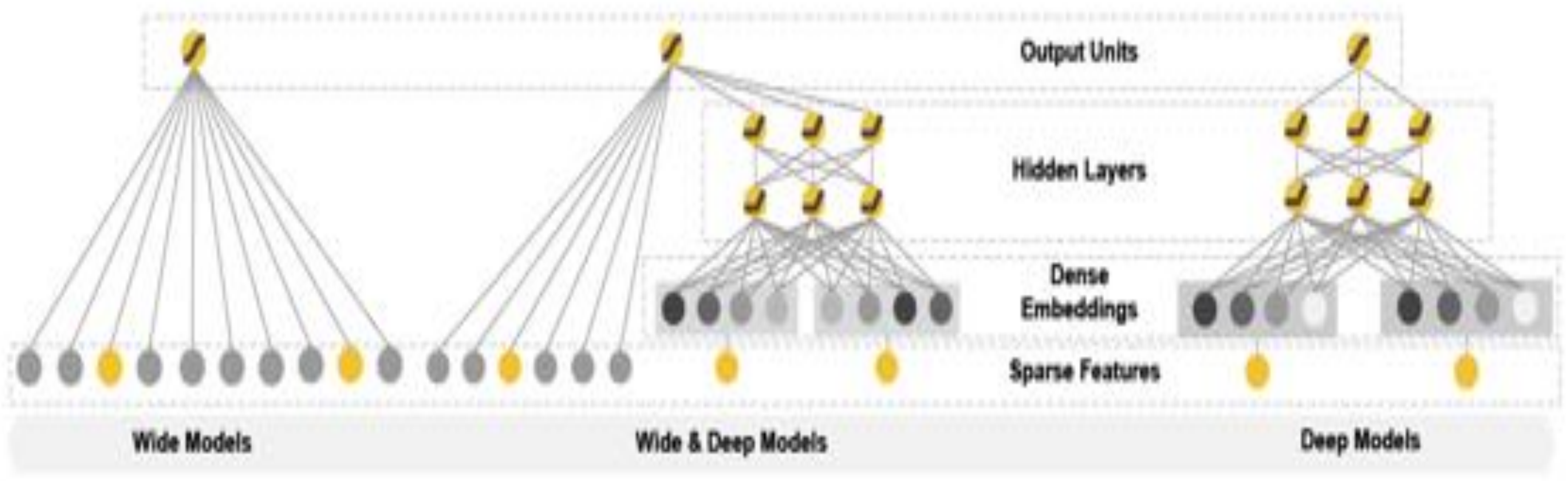
Ensemble Learning

Contrast: "Wide" vs "Deep" models (see "Wide & Deep Learning for Recommender Systems" by Cheng et al.)



Ensemble Learning

Contrast: "Wide" vs "Deep" models (see "Wide & Deep Learning for Recommender Systems" by Cheng et al.)



Deep networks good for global effects (in input feature space), wide networks better for local effects.

Gradient Boosting

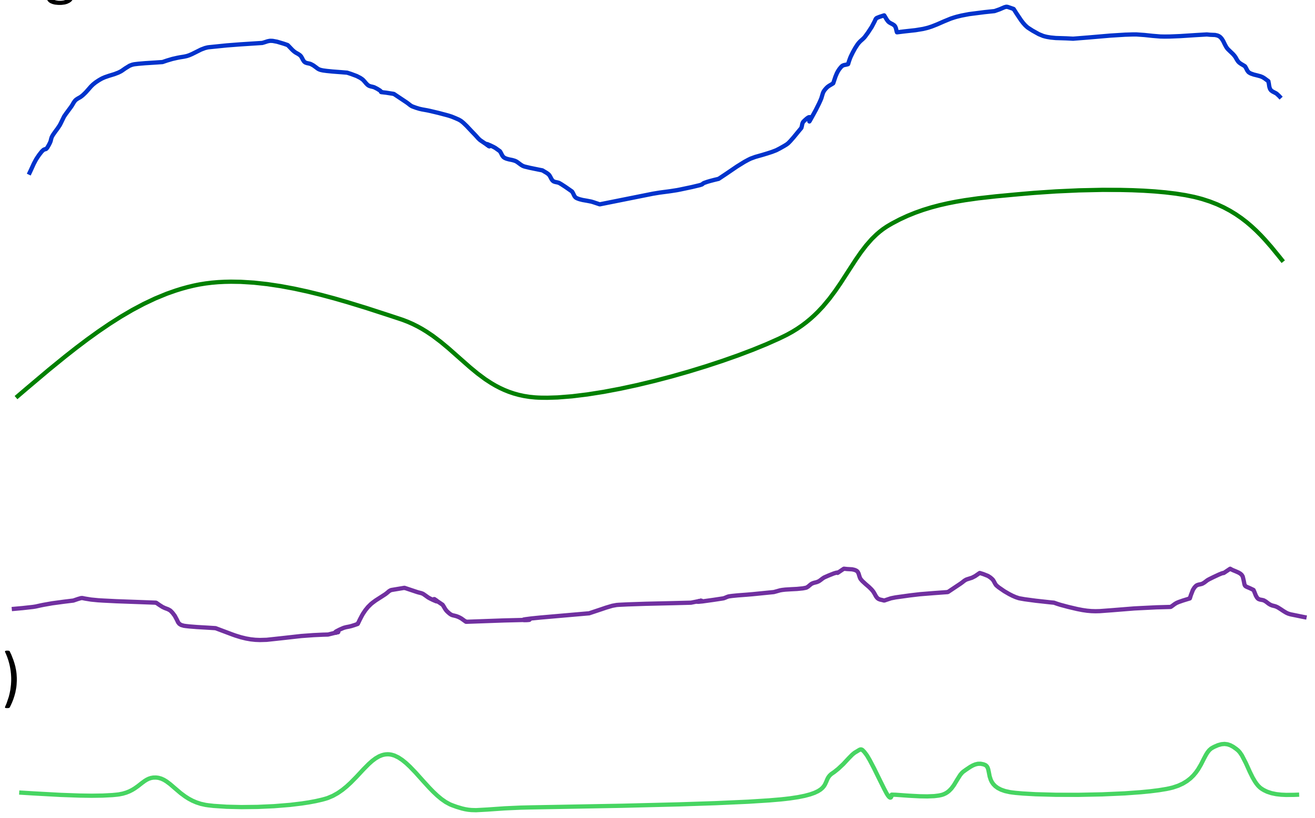
Gradient boosting:

Input

First model

First residual
(input – model1)

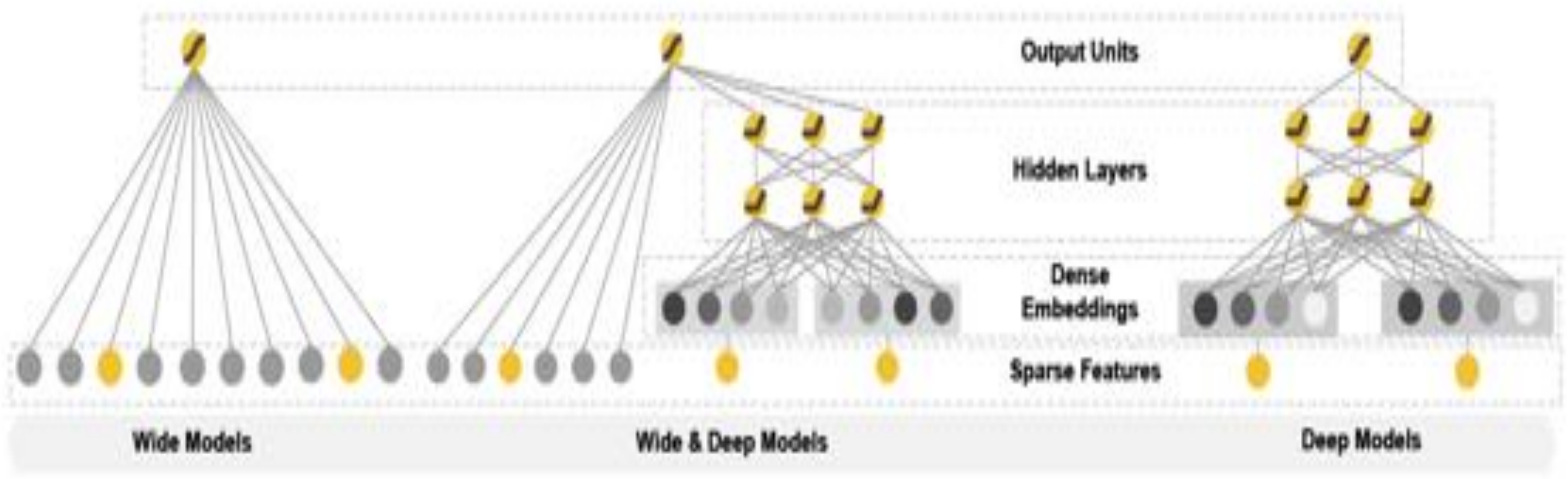
Second model



Models progress from global → local

Ensemble Learning

Contrast: “Wide” vs “Deep” models (see “Wide & Deep Learning for Recommender Systems” by Cheng et al.)



Deep networks good for global effects (in input feature space), wide networks better for local effects.

In boosting, the base learners model global → local effects.

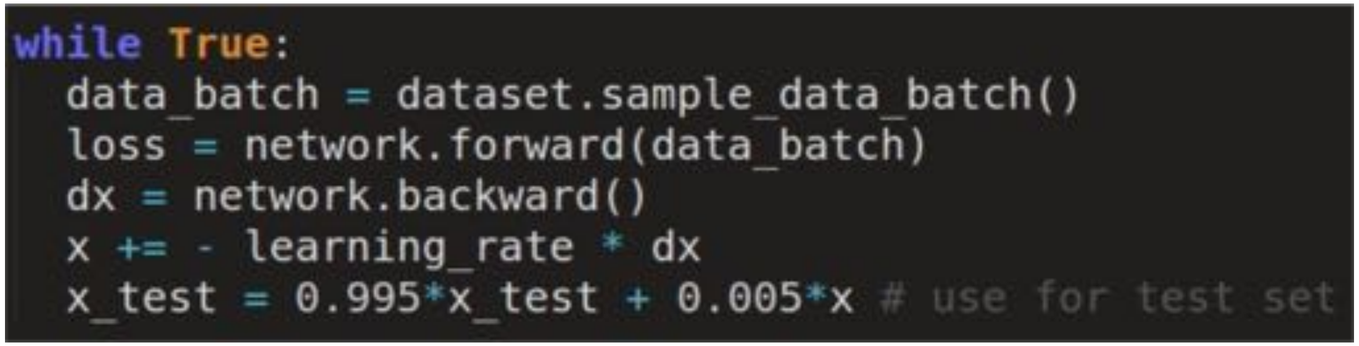
So boost with deep → wide networks? - Open Question!

Fun Tips/Tricks:

- can also get a small boost from averaging multiple model checkpoints of a single model.

Fun Tips/Tricks:

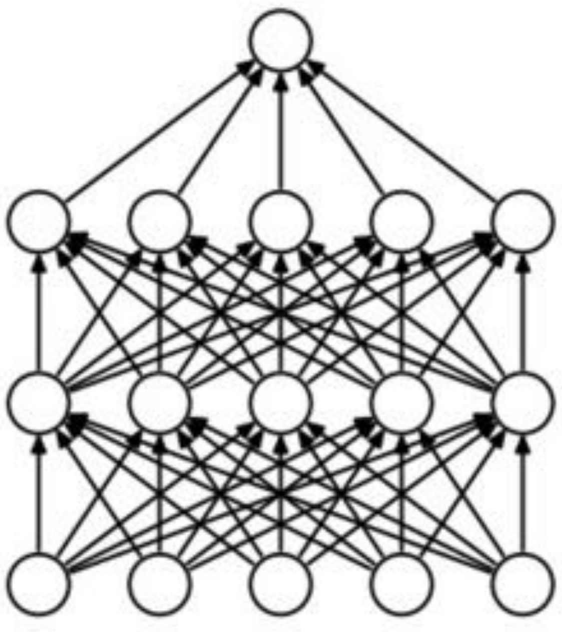
- can also get a small boost from averaging multiple model checkpoints of a single model.
- keep track of (and use at test time) a running average parameter vector:

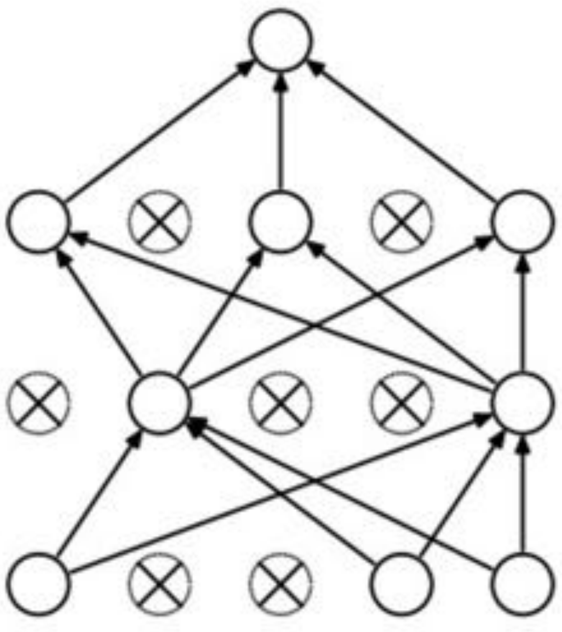

```
while True:
  data_batch = dataset.sample_data_batch()
  loss = network.forward(data_batch)
  dx = network.backward()
  x += - learning_rate * dx
  x_test = 0.995*x_test + 0.005*x # use for test set
```

Regularization (dropout)

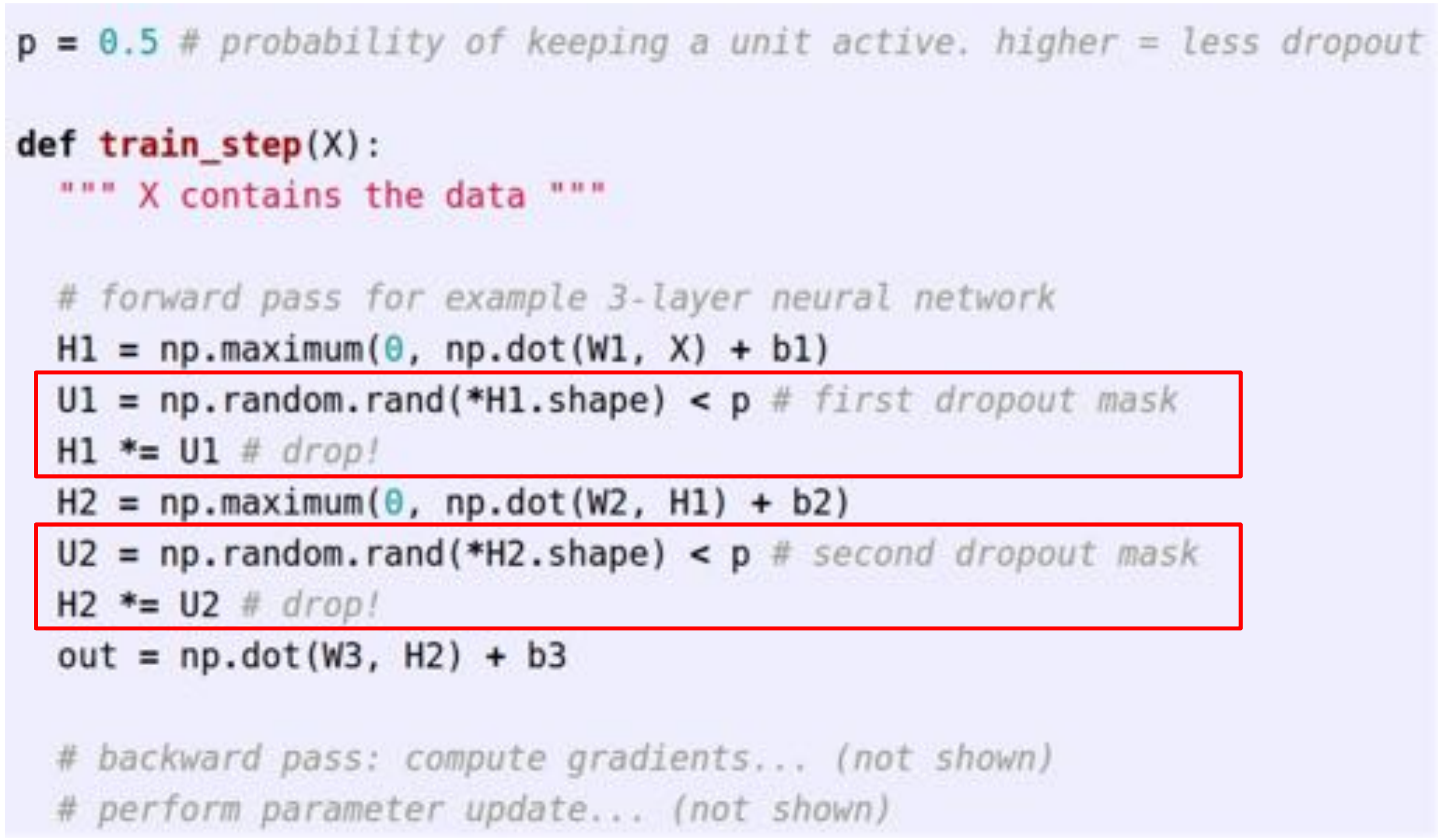
Regularization by Dropout

“randomly set some neurons to zero in the forward pass”


(a) Standard Neural Net

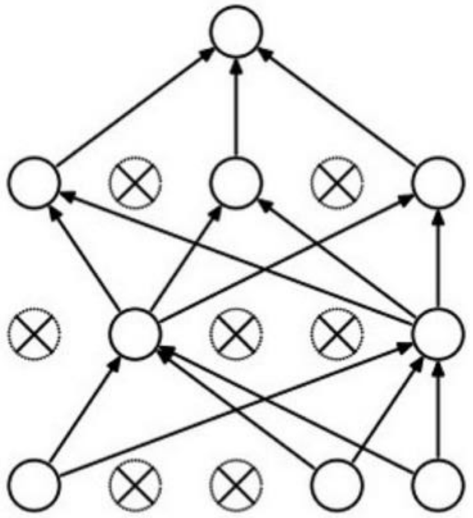

(b) After applying dropout.

[Srivastava et al., 2014]

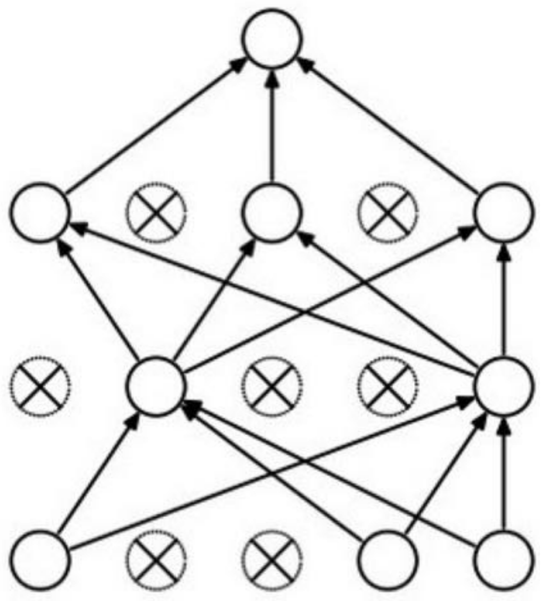


```
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)
```

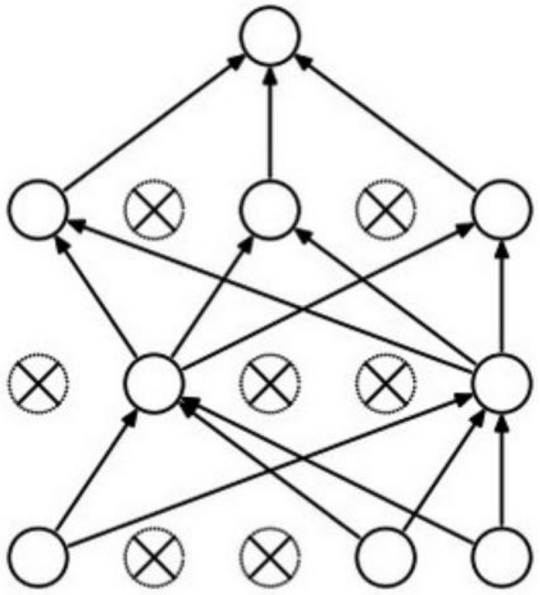
Example forward pass with a 3-layer network using dropout



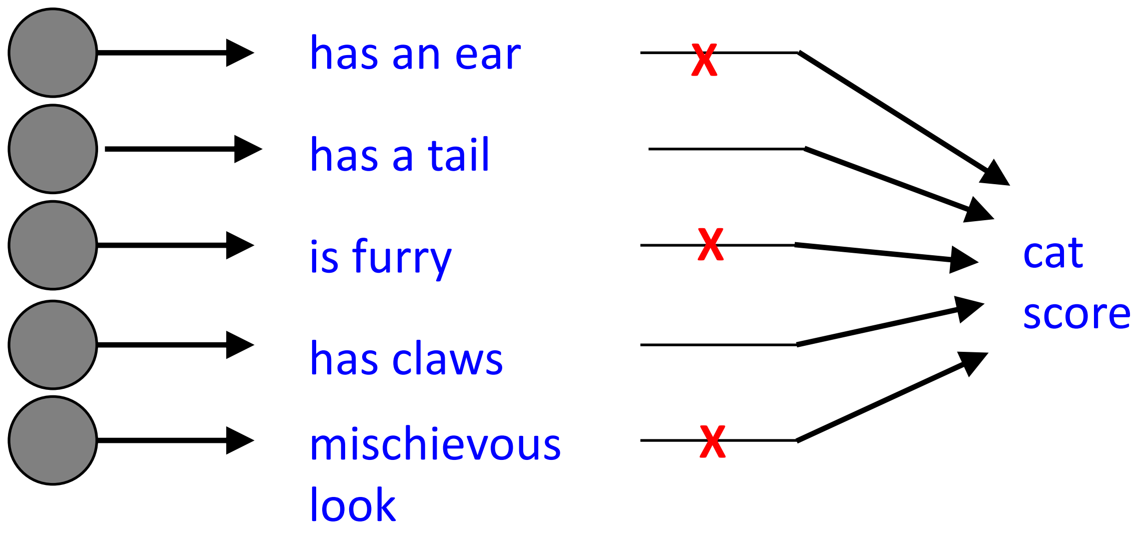
Waaaait a second…
How could this possibly be a good idea?



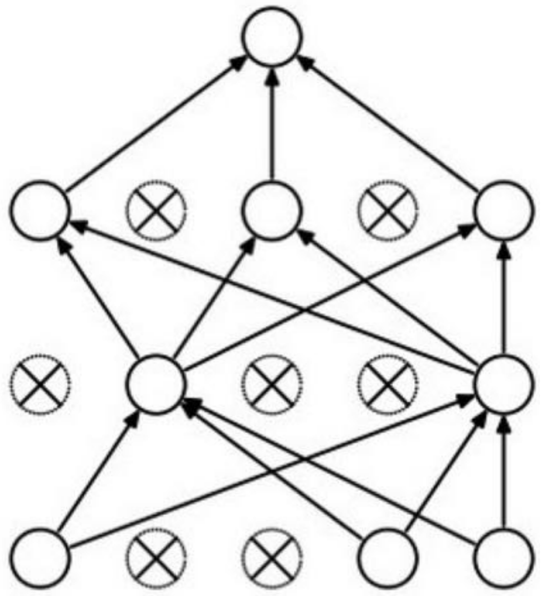
Waaaait a second…
How could this possibly be a good idea?



Forces the network to have a redundant representation.



Waaaait a second…
How could this possibly be a good idea?

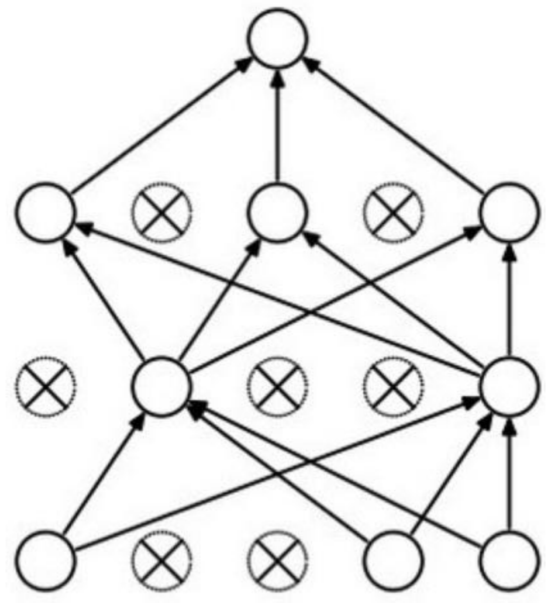


Another interpretation:

Dropout is training a large ensemble of models (that share parameters).

Each binary mask is one model, gets trained on only ~one datapoint.

At test time….



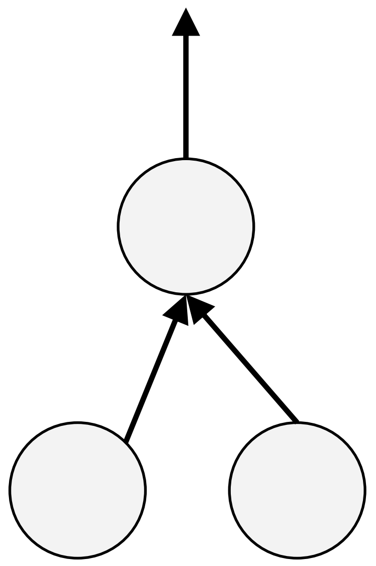
Ideally:
want to integrate out all the noise

Monte Carlo approximation:
do many forward passes with different dropout masks, average all predictions

At test time….

Can in fact do this with a single forward pass! (approximately)

Leave all input neurons turned on (no dropout).

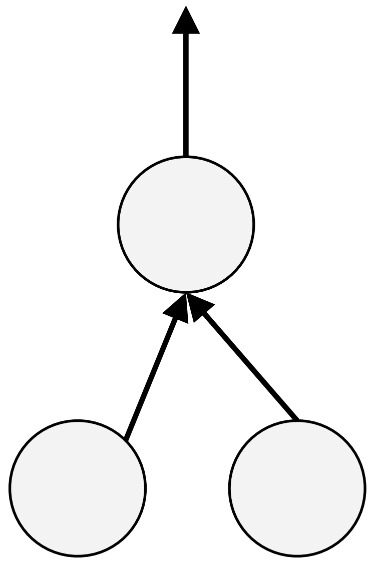


(this can be shown to be an approximation to evaluating the whole ensemble)

At test time….

Can in fact do this with a single forward pass! (approximately)

Leave all input neurons turned on (no dropout).



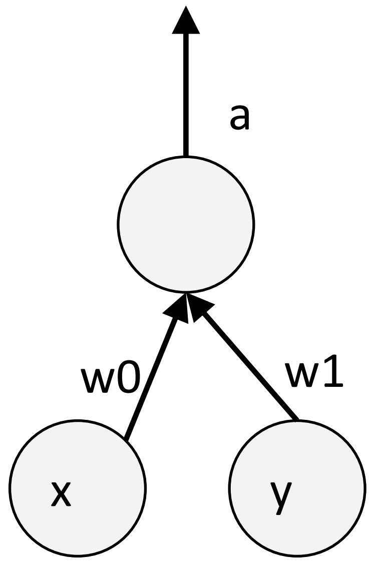
Q: Suppose that with all inputs present at test time the output of this neuron is x.

What would its output be during training time, in expectation? (e.g. if p = 0.5)

At test time….

Can in fact do this with a single forward pass! (approximately)

Leave all input neurons turned on (no dropout).



during test: **a = w0*x + w1*y**

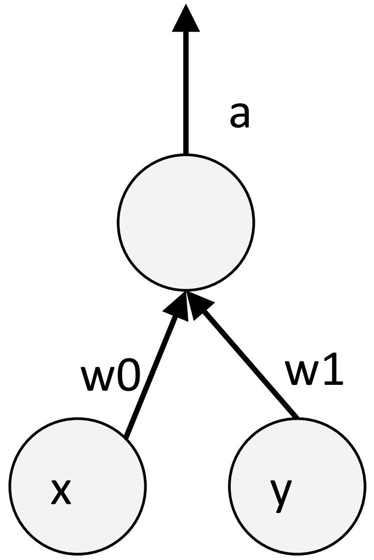
during train:

E[a] = ¼ * (w0*0 + w1*0
w0*0 + w1*y
w0*x + w1*0
w0*x + w1*y)
= ¼ * (2 w0*x + 2 w1*y)
= **½ * (w0*x + w1*y)**

At test time….

Can in fact do this with a single forward pass! (approximately)

Leave all input neurons turned on (no dropout).



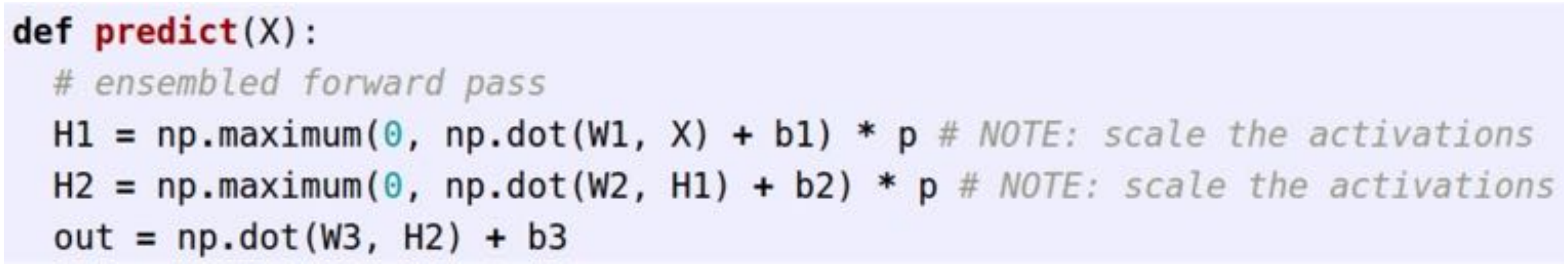
during test: **a = w0*x + w1*y**

during train:

E[a] = ¼ * (w0*0 + w1*0
w0*0 + w1*y
w0*x + w1*0
w0*x + w1*y)
= ¼ * (2 w0*x + 2 w1*y)
= **½ * (w0*x + w1*y)**

With p=0.5, using all inputs in the forward pass would inflate the activations by 2x from what the network was "used to" during training!
=> Have to compensate by scaling the activations back down by ½

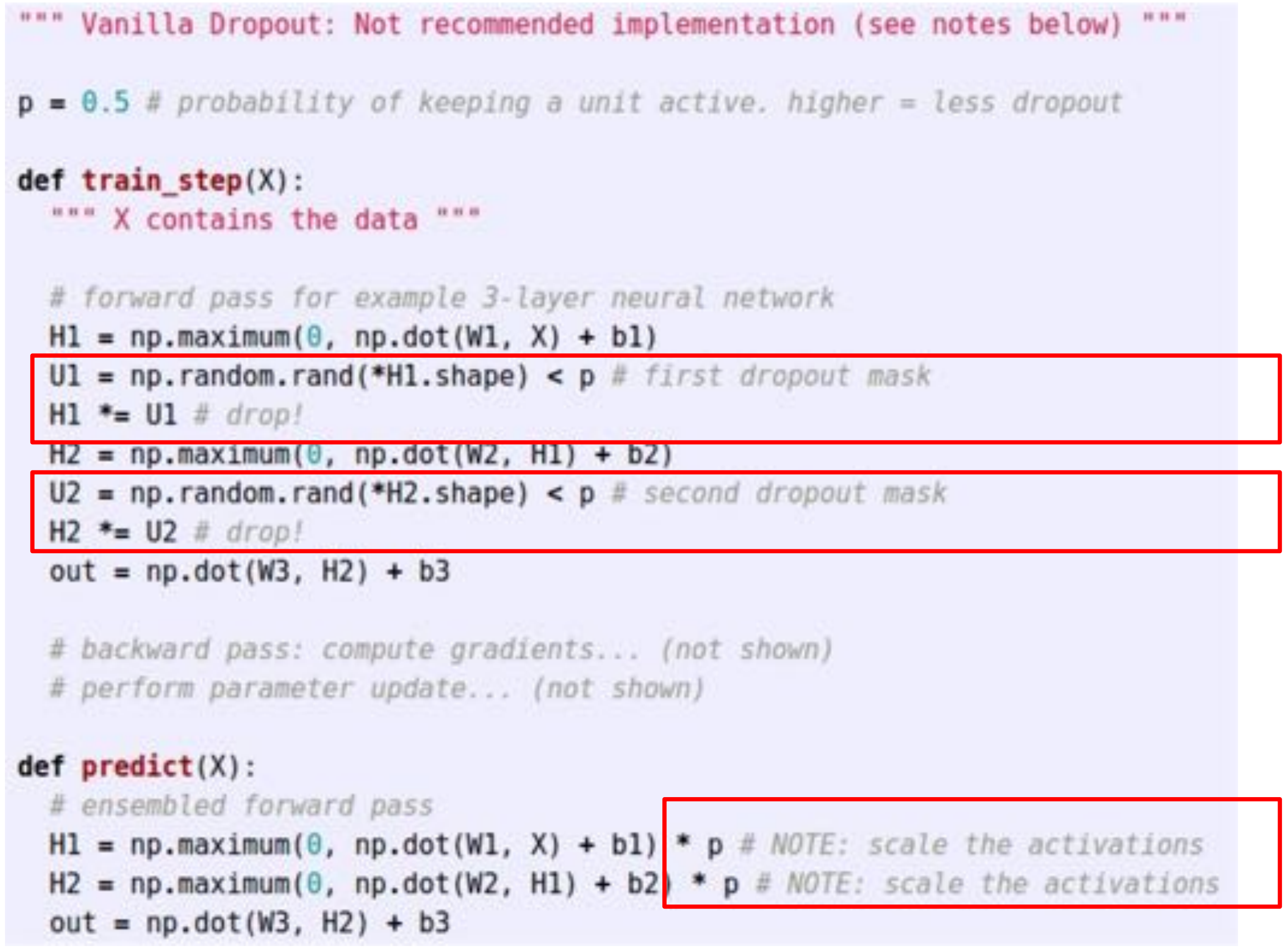
We can do something approximate analytically



```
def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
  H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
  out = np.dot(W3, H2) + b3
```

At test time all neurons are active always
=> We must scale the activations so that for each neuron:
<u>output at test time</u> = <u>expected output at training time</u>

Dropout Summary

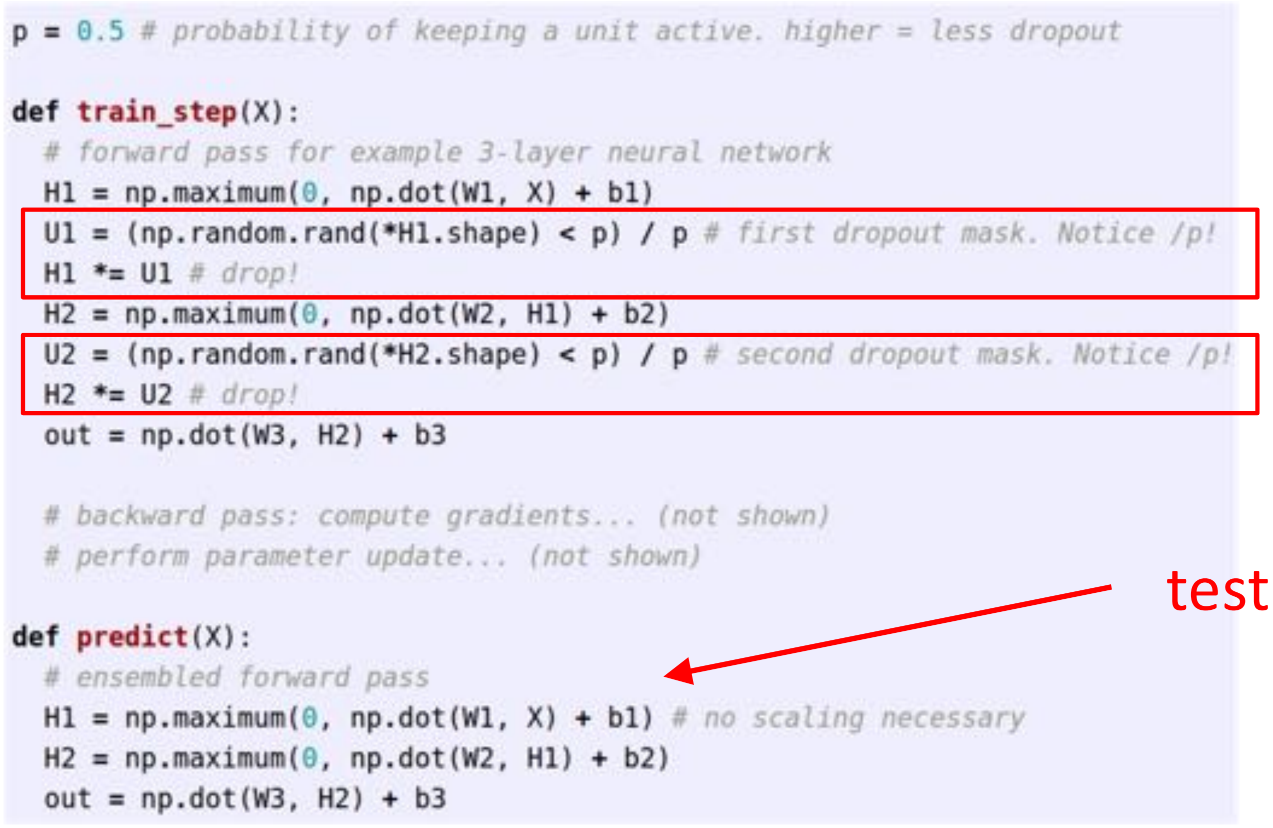


```
""" Vanilla Dropout: Not recommended implementation (see notes below) """

p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
  H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
  out = np.dot(W3, H2) + b3
```

drop in forward pass

scale at test time

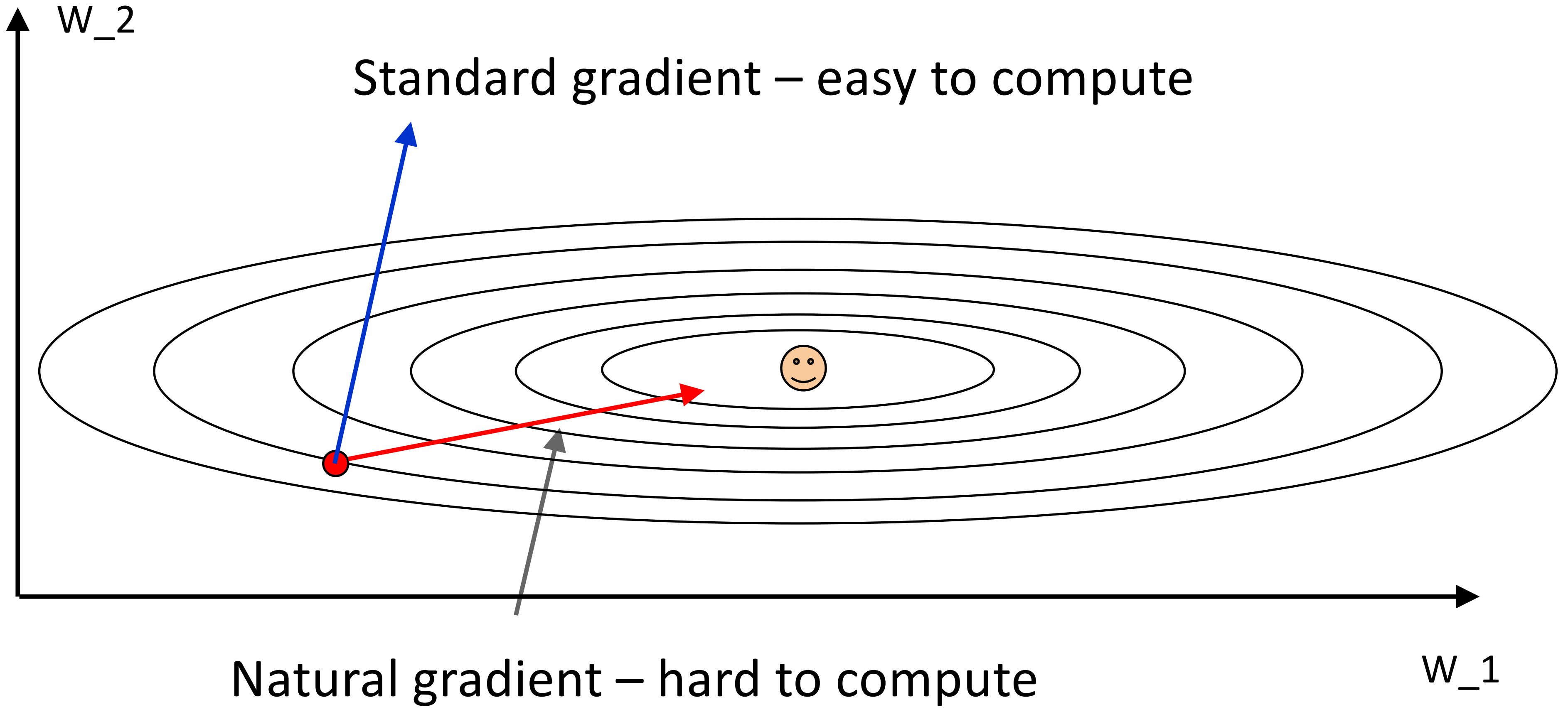
More common: “Inverted dropout”



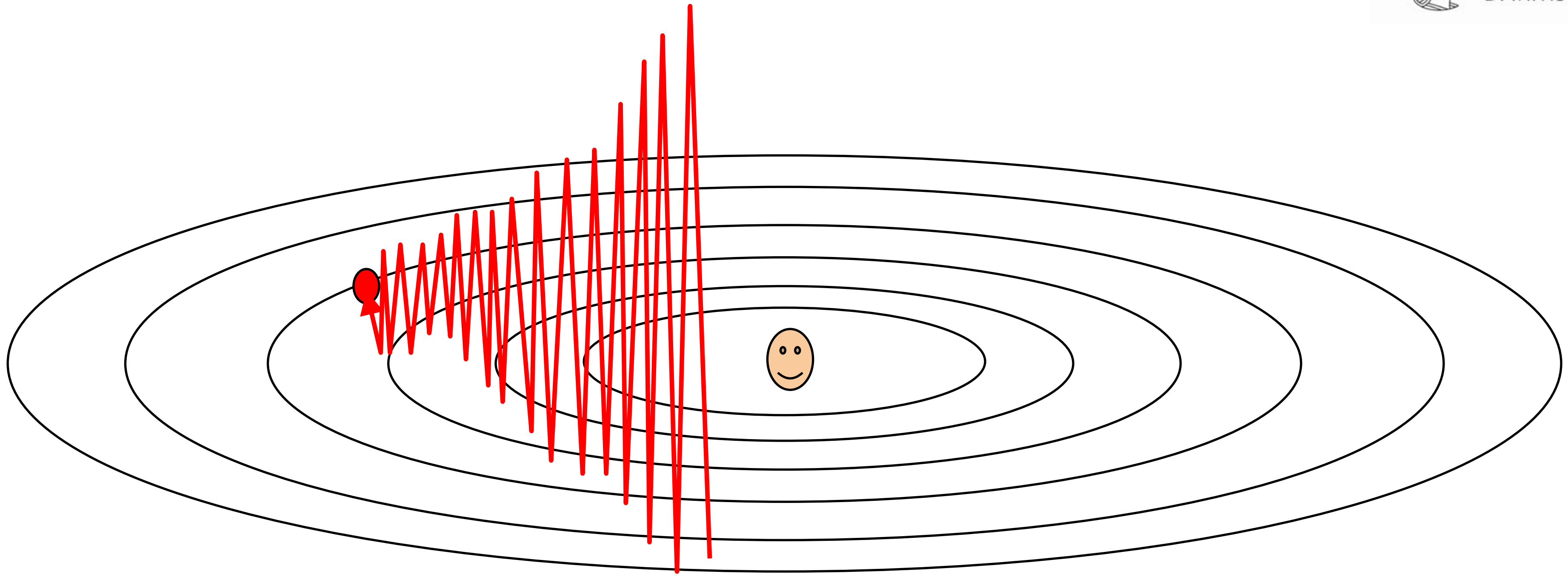
```
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  out = np.dot(W3, H2) + b3
```

test time is unchanged!

Back to gradients


W_2
Standard gradient – easy to compute
Natural gradient – hard to compute
W_1

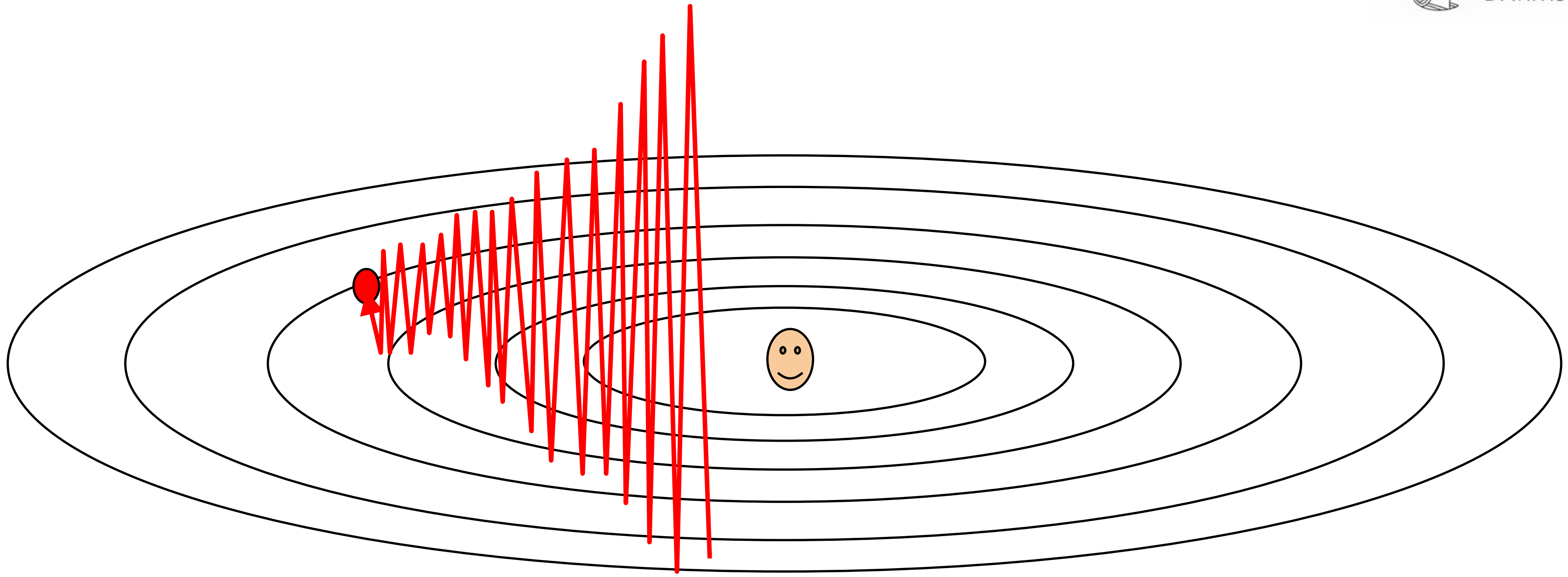
Gradient Magnitudes:



Gradients too big → divergence

Gradients too small → slow convergence

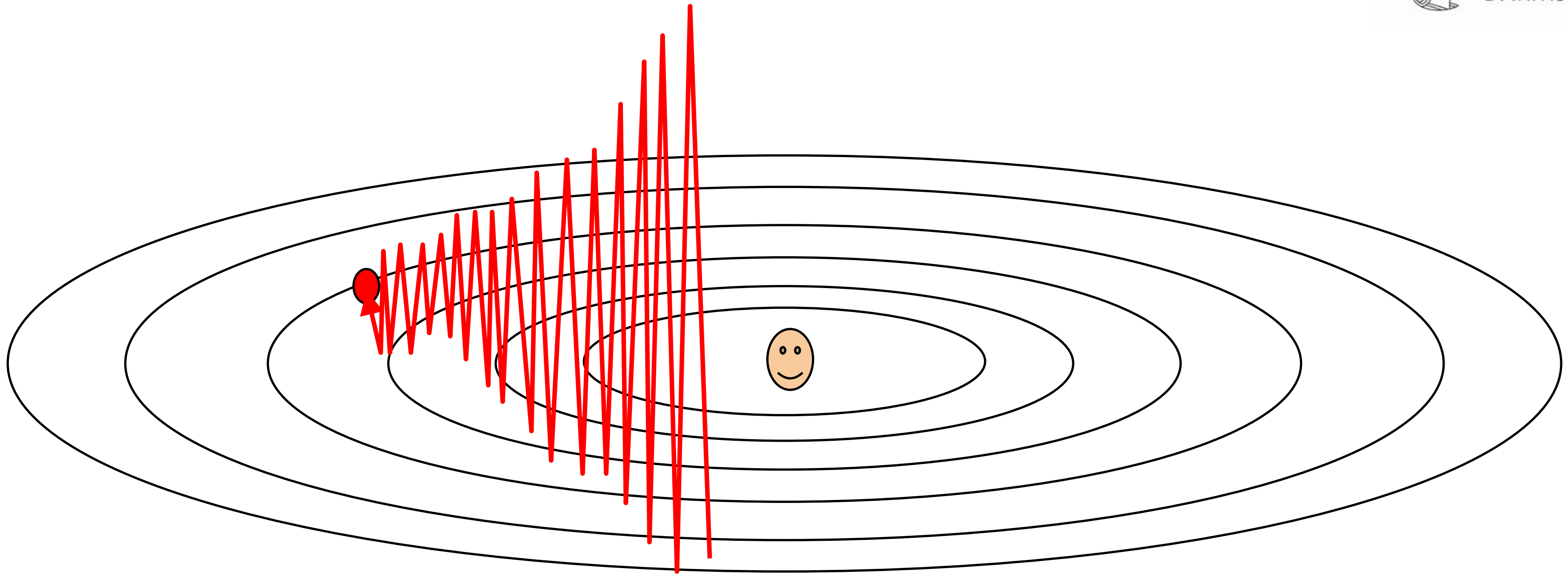
Gradient Magnitudes:



Gradients too big → divergence
Gradients too small → slow convergence

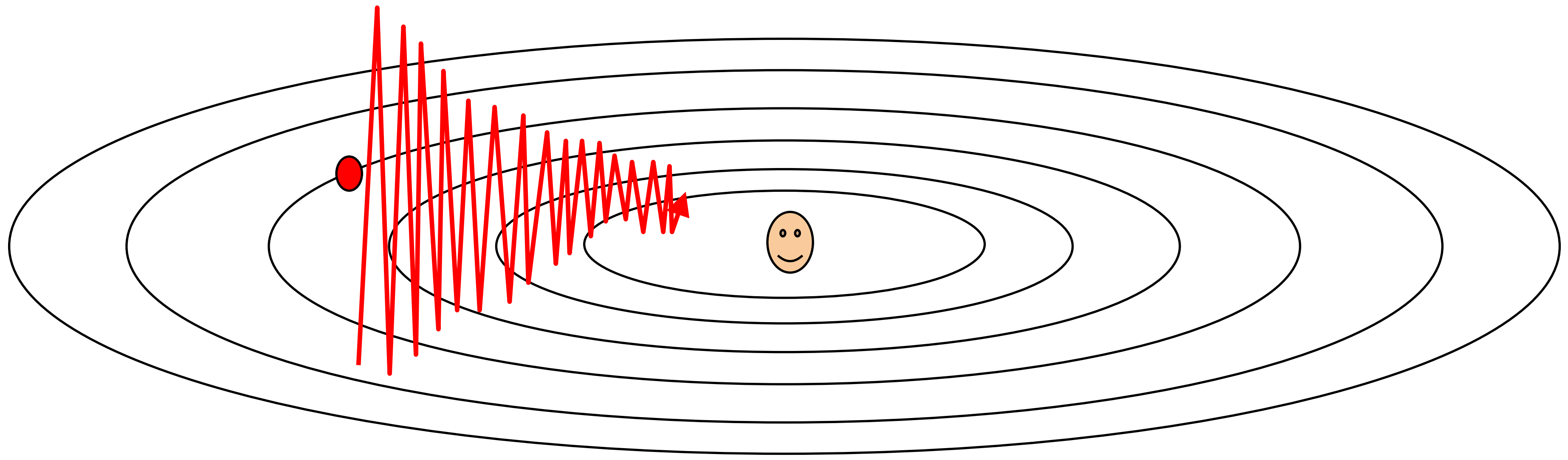
Divergence is much worse!

Gradient Magnitudes:



What's the simplest way to ensure gradients stay bounded?

Gradient clipping:

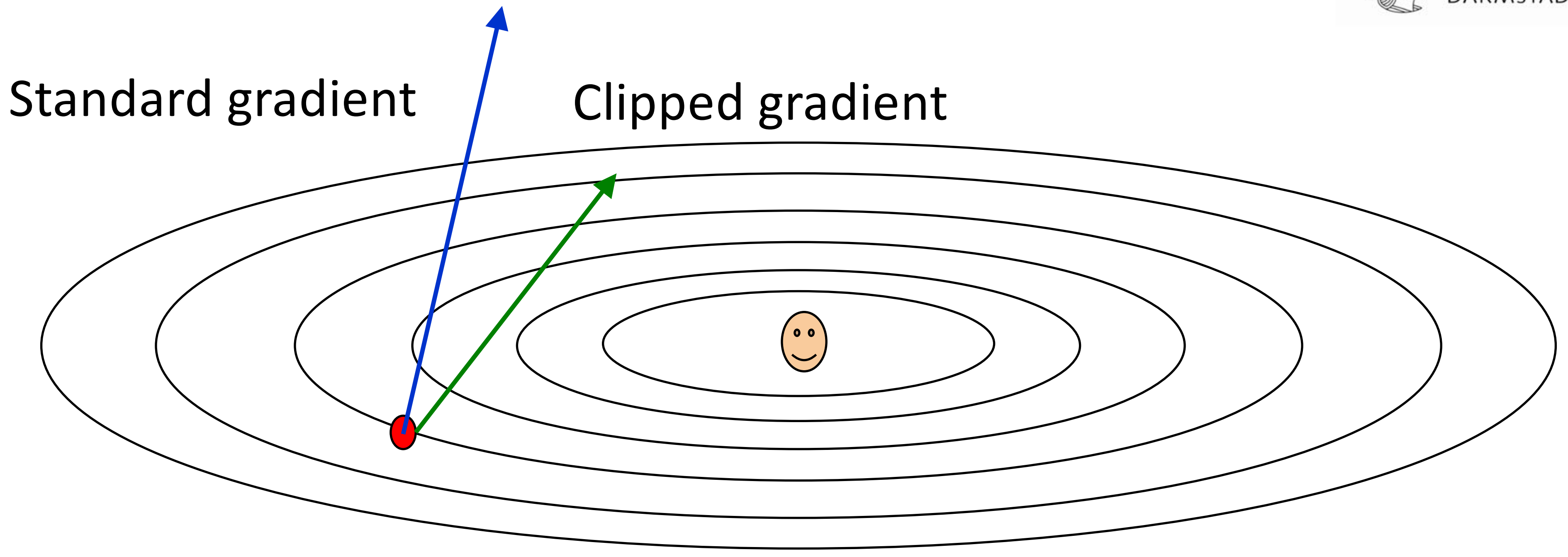


Simply limit the magnitude of each gradient:

$$\bar{g}_i = \min(g_{\max}, \max(-g_{\max}, g_i))$$

so $|\bar{g}_i| \leq g_{\max}$. Then use a decreasing learning rate to converge to an optimum.

Extreme Gradient clipping:

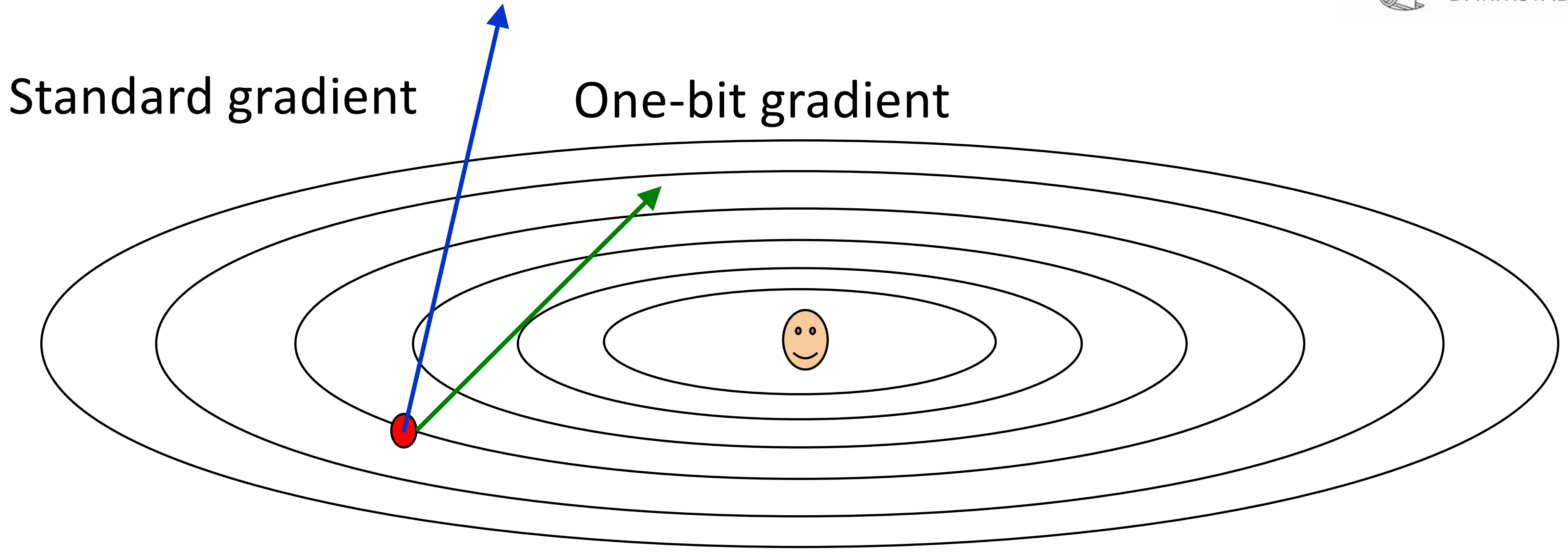


Gradient clipping limits the largest gradient dimensions, while others may be very small.

ADAGRAD and RMSprop scale gradient dimensions by the inverse std deviation, so all dimension have unit sdev.

What if we scale gradients up before clipping, so all dimensions are clipped?

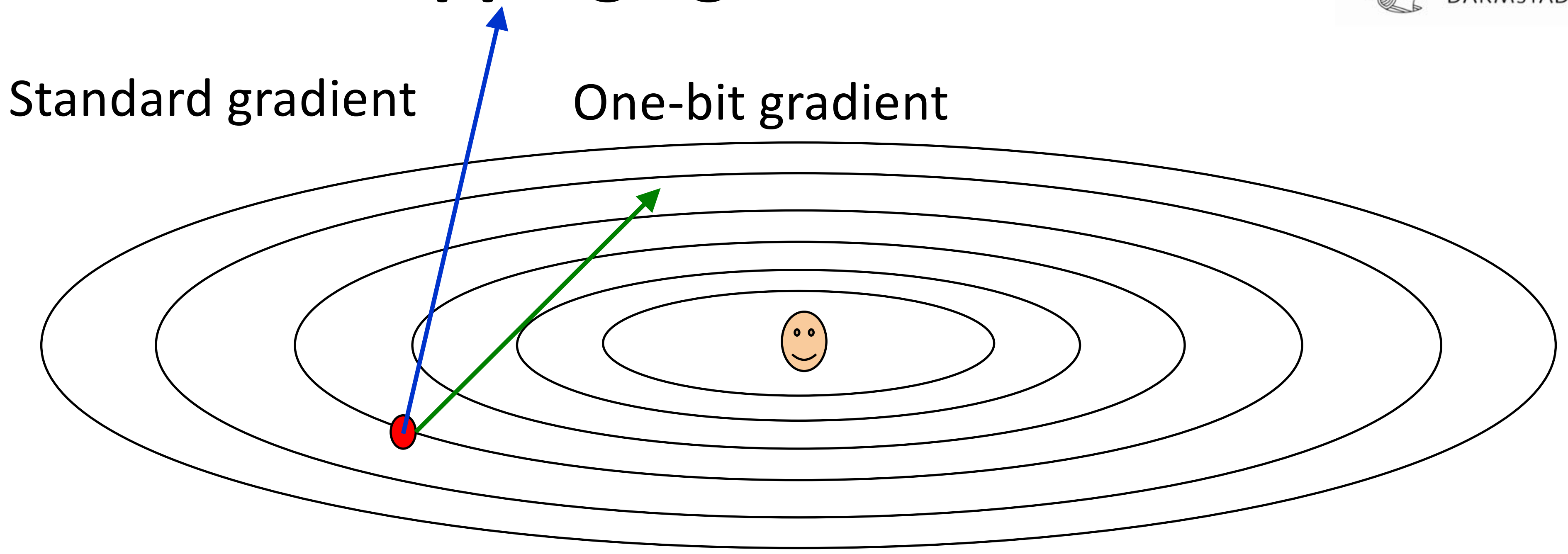
One-bit gradients



If we clip all gradient dimensions, we are left only with their sign: $\bar{g}_i = g_{\max}(-1,1,1,-1,1,\ldots)$

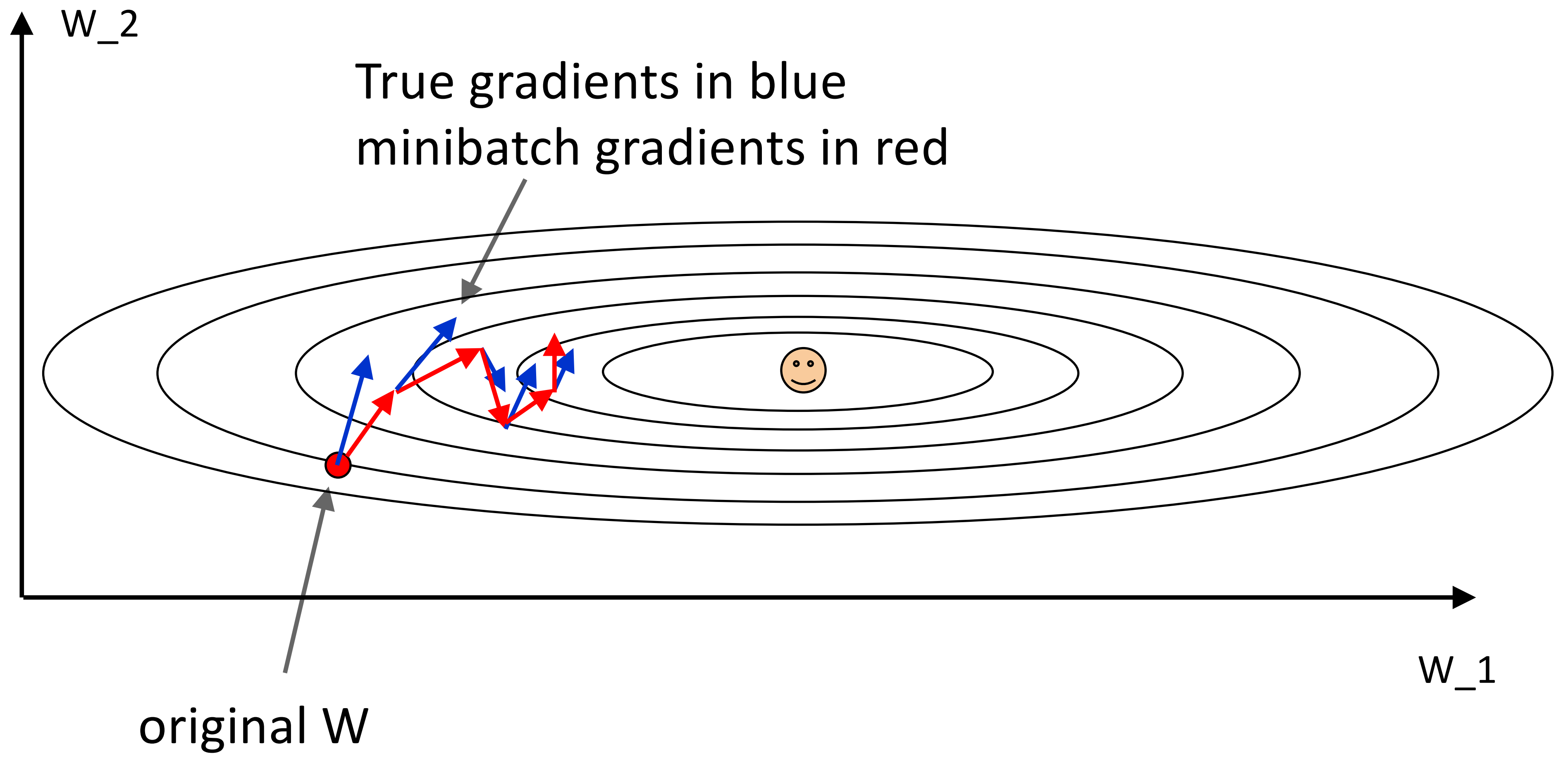
This actually works on some problems with little or no loss of accuracy: (see “1-Bit Stochastic Gradient Descent and Application to Data-Parallel Distributed Training of Speech DNNs” by Seide et al. 2014)

Gradient Clipping again



Though more commonly gradient clipping is used less aggressively (don’t saturate all dimensions) as a option to other algorithms like ADAGRAD, RMSprop.

Stochastic Gradient Descent



Gradients are noisy but still make good progress on average

Gradient Noise

If a little noise is good, what about **adding** noise to gradients?

Gradient Noise

If a little noise is good, what about **adding** noise to gradients?

A: Works Great for many models!

Is especially valuable for complex models that would overfit otherwise.

“Adding Gradient Noise Improves Learning for Very Deep Networks” Arvind Neelakantan et al., 2016

Gradient Noise

Schedule:

$$g_t \leftarrow g_t + N(0, \sigma_t^2)$$

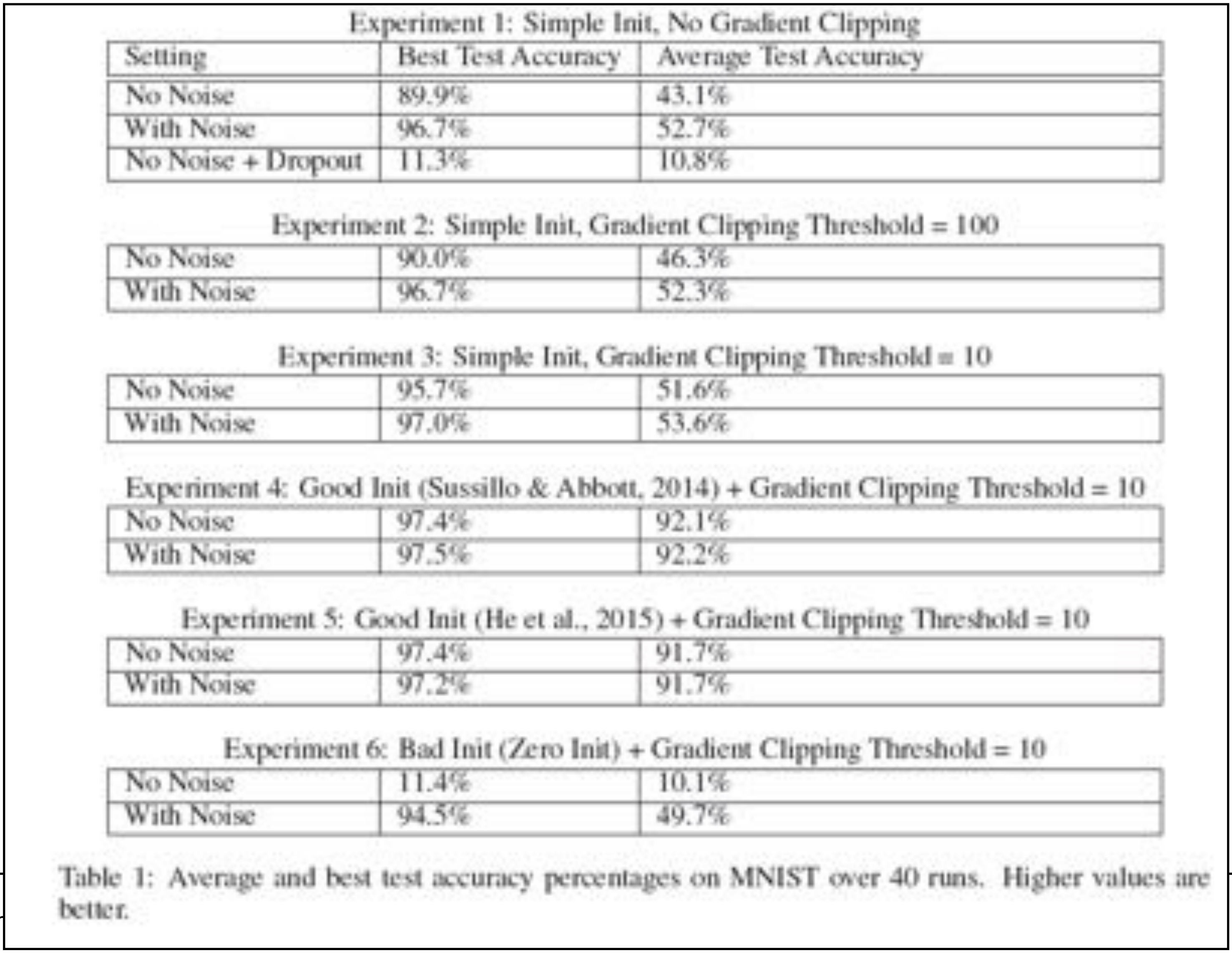
where the noise variance is:

$$\sigma_t^2 = \frac{\eta}{(1+t)^\gamma}$$

with η selected from $\{0.01, 0.3, 1.0\}$ and $\gamma = 0.55$.

Gradient Noise

Results on MNIST with a 20-layer ReLU network:



Experiment 1: Simple Init, No Gradient Clipping

Setting	Best Test Accuracy	Average Test Accuracy
No Noise	89.9%	43.1%
With Noise	96.7%	52.7%
No Noise + Dropout	11.3%	10.8%

Experiment 2: Simple Init, Gradient Clipping Threshold = 100

No Noise	90.0%	46.3%
With Noise	96.7%	52.3%

Experiment 3: Simple Init, Gradient Clipping Threshold = 10

No Noise	95.7%	51.6%
With Noise	97.0%	53.6%

Experiment 4: Good Init (Sussillo & Abbott, 2014) + Gradient Clipping Threshold = 10

No Noise	97.4%	92.1%
With Noise	97.5%	92.2%

Experiment 5: Good Init (He et al., 2015) + Gradient Clipping Threshold = 10

No Noise	97.4%	91.7%
With Noise	97.2%	91.7%

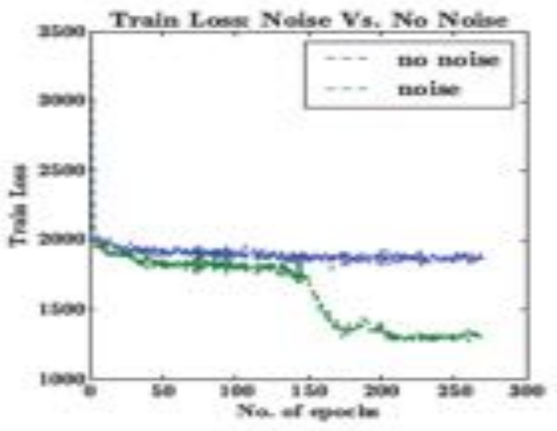
Experiment 6: Bad Init (Zero Init) + Gradient Clipping Threshold = 10

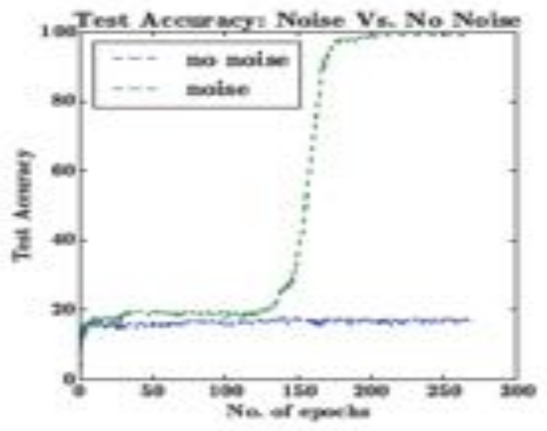
No Noise	11.4%	10.1%
With Noise	94.5%	49.7%

Table 1: Average and best test accuracy percentages on MNIST over 40 runs. Higher values are better.

Gradient Noise

Neural Programmer:





Neural RAM: (learning a search task)

	Hyperparameter-1	Hyperparameter-2	Hyperparameter-3	Average
No Noise	1%	0%	3%	1.3%
With Noise	5%	22%	7%	11.3%

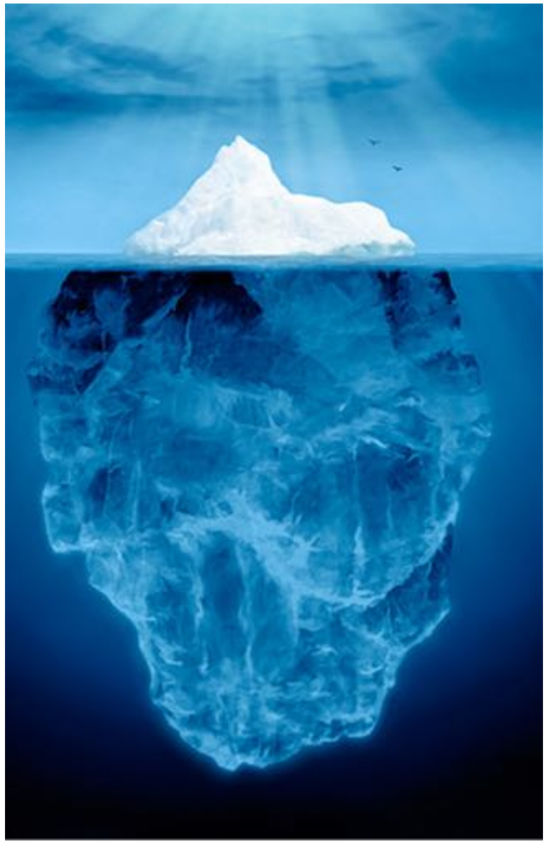
Neural GPUs: (learning arithmetic operations)

Setting	Error $< 1\%$	Error $< 2\%$	Error $< 3\%$	Error $< 5\%$
No Noise	28	90	172	387
With Noise	58	159	282	570

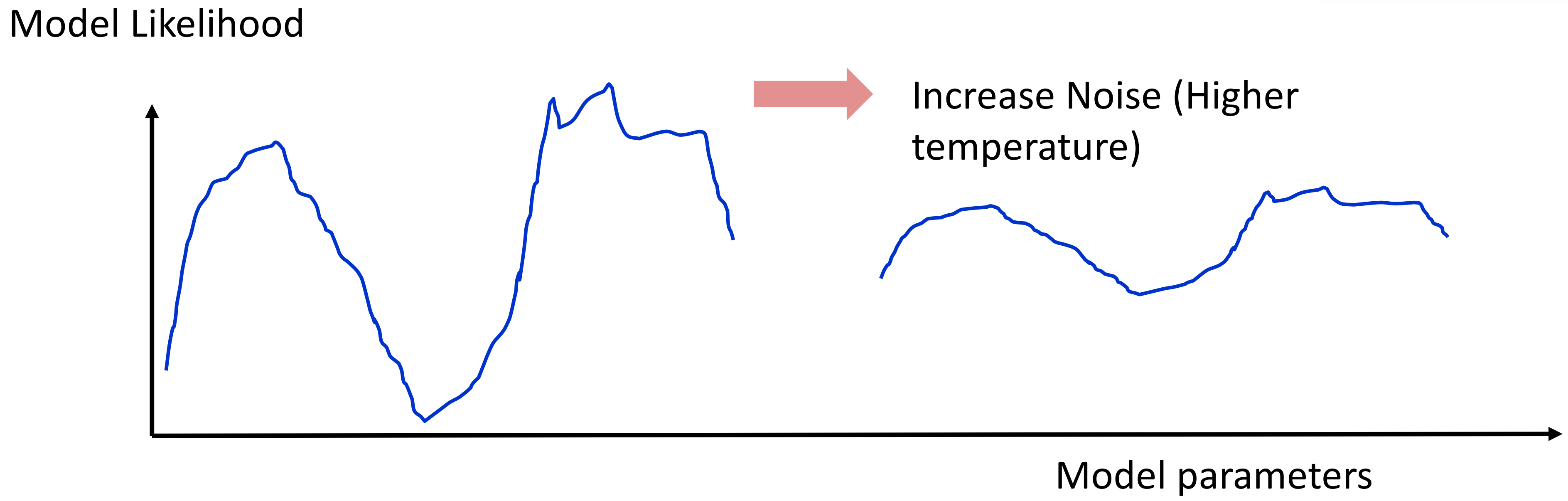
Gradient Noise

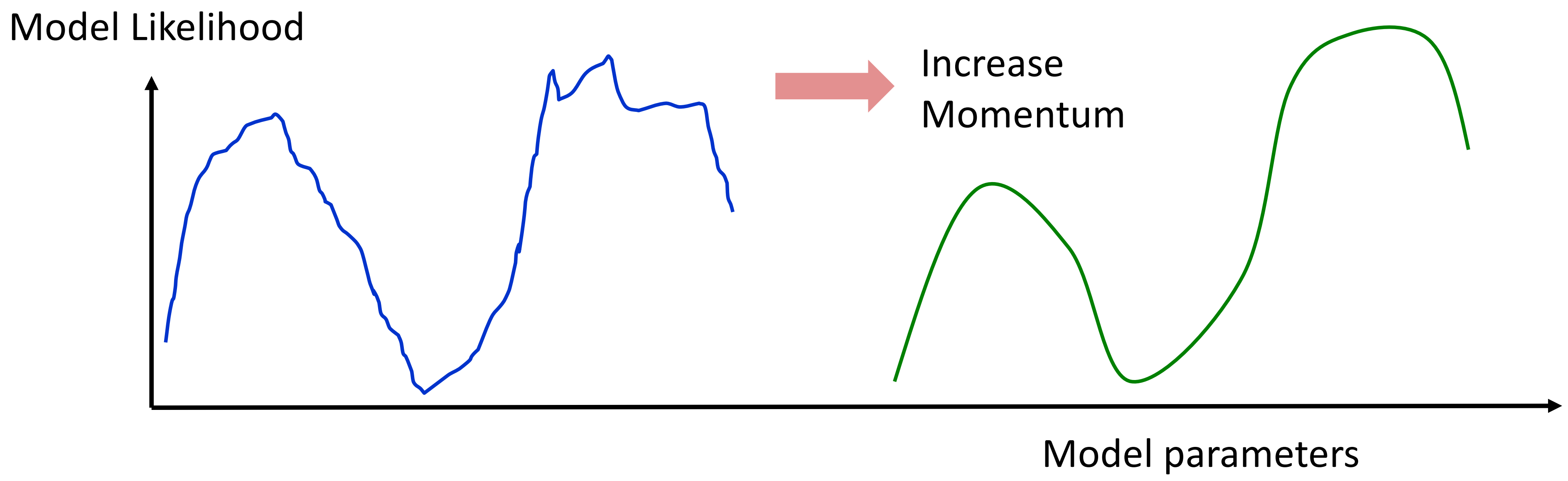
Very interesting questions here:

- Gradient noise turns model parameter into a Bayesian inference task:
- Noise magnitude controls “flatness” of the distribution.
- Adding momentum controls level-of-detail.



Gradient Noise + Momentum





What have we learnt?

- Hyperparameter optimization
- Ensembles
- Dropout
- One-bit gradients
- Gradient noise