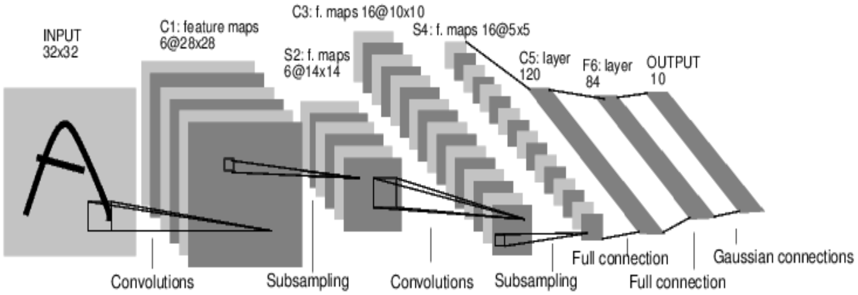
Designing, Visualizing and Understanding Deep Neural Networks

Lecture 7: Training Networks

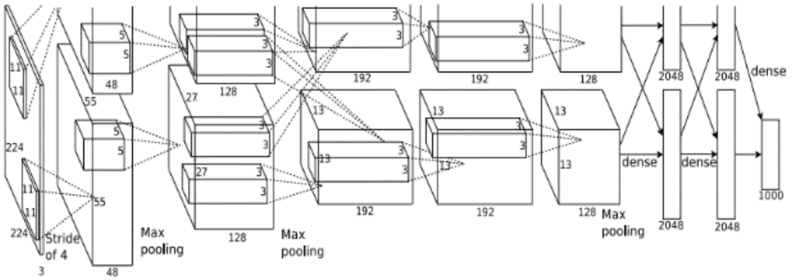
CS 182/282A Spring 2019
John Canny

Last Time: CNN Case Studies:

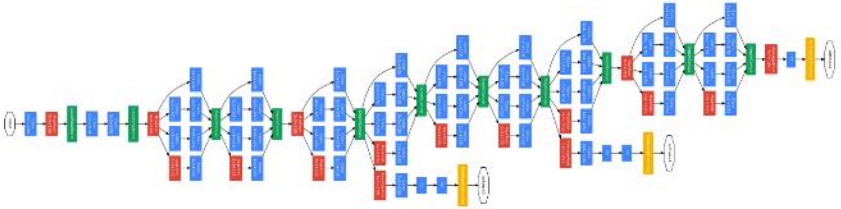
LeNet 5 [*LeCun et al. 98*]



AlexNet [*Krizhevsky et al. 2012*]



Inception/GoogLeNet [*Szegedy et al., 2014*]



Case Study: ResNet [*He et al., 2015*]

Last Time: Transfer Learning with CNNs

1. Train on Imagenet

2. If small dataset: fix all weights (treat CNN as fixed feature extractor), retrain only the classifier

i.e. swap the Softmax layer at the end

3. If you have medium sized dataset, **"finetune"** instead: use the old weights as initialization, train the full network or only some of the higher layers

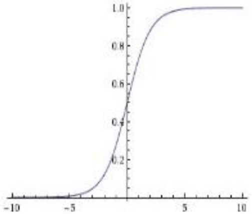
retrain bigger portion of the network, or even all of it.

tip: use only ~1/10th of the original learning rate in finetuning to player, and ~1/100th on intermediate layers

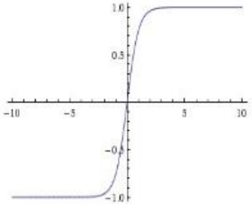
Last Time: Activation Functions

Sigmoid

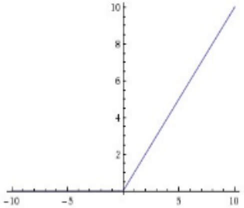
$\sigma(x) = 1/(1+e^{-x})$



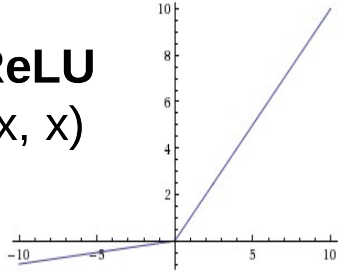
tanh tanh(x)



ReLU max(0,x)

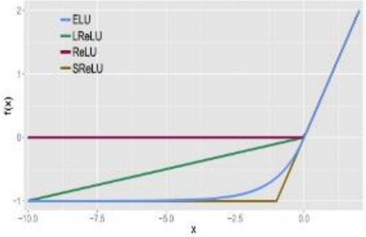


Leaky ReLU
max(0.1x, x)



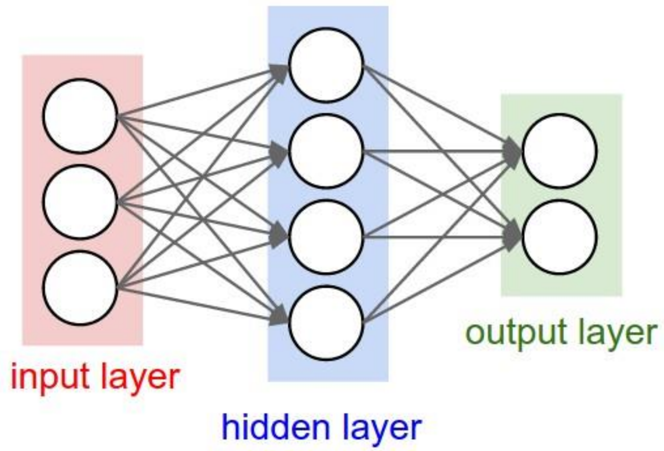
Maxout $\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU $f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\,(\exp(x) - 1) & \text{if } x \le 0 \end{cases}$



Last Time: Weight Initialization

Xavier initialization avoids vanishing or exploding gradients, simulates random input activations and random weights.



Reminders

- Assignment 1 is due 2/19 (Tuesday)
- Project proposal due 2/20 (Wednesday)

This Time: Batch Normalization [Ioffe and Szegedy, 2015]

“you want unit gaussian activations? just make them so.”

Consider a batch of activations at some layer.
To make each dimension unit gaussian, apply:

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

this is a vanilla differentiable function...

This Time: Batch Normalization

[Ioffe and Szegedy, 2015]

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$\boxed{y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}}$$

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Reduces need for dropout

Un-normalization!! Re-compute and apply the optimal scaling and bias for each neuron!
Learn γ and β (same dims as μ and σ^2).
It can (should?) learn the identity mapping!

Batch Normalization Gradients

[Ioffe and Szegedy, 2015]

$$\frac{\partial \ell}{\partial \widehat{x}_i} = \frac{\partial \ell}{\partial y_i} \cdot \gamma$$

$$\frac{\partial \ell}{\partial \sigma_\mathcal{B}^2} = \sum_{i=1}^m \frac{\partial \ell}{\partial \widehat{x}_i} \cdot (x_i - \mu_\mathcal{B}) \cdot \frac{-1}{2}(\sigma_\mathcal{B}^2 + \epsilon)^{-3/2}$$

$$\frac{\partial \ell}{\partial \mu_\mathcal{B}} = \left(\sum_{i=1}^m \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}} \right) + \frac{\partial \ell}{\partial \sigma_\mathcal{B}^2} \cdot \frac{\sum_{i=1}^m -2(x_i - \mu_\mathcal{B})}{m}$$

Don't need these directly, they are subexpressions for the other gradients.

Think of this as backprop for the nodes $\hat{x}$, $\sigma_\mathcal{B}^2$, $\mu_\mathcal{B}$, which are all internal to the minibatch update.

$$\frac{\partial \ell}{\partial x_i} = \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{1}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}} + \frac{\partial \ell}{\partial \sigma_\mathcal{B}^2} \cdot \frac{2(x_i - \mu_\mathcal{B})}{m} + \frac{\partial \ell}{\partial \mu_\mathcal{B}} \cdot \frac{1}{m}$$

$$\frac{\partial \ell}{\partial \gamma} = \sum_{i=1}^m \frac{\partial \ell}{\partial y_i} \cdot \widehat{x}_i$$

$$\frac{\partial \ell}{\partial \beta} = \sum_{i=1}^m \frac{\partial \ell}{\partial y_i}$$

Batch Normalization Gradients

[Ioffe and Szegedy, 2015]

$$\frac{\partial \ell}{\partial \widehat{x}_i} = \frac{\partial \ell}{\partial y_i} \cdot \gamma$$

$$\frac{\partial \ell}{\partial \sigma_\mathcal{B}^2} = \sum_{i=1}^m \frac{\partial \ell}{\partial \widehat{x}_i} \cdot (x_i - \mu_\mathcal{B}) \cdot \frac{-1}{2}(\sigma_\mathcal{B}^2 + \epsilon)^{-3/2}$$

$$\frac{\partial \ell}{\partial \mu_\mathcal{B}} = \left(\sum_{i=1}^m \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}} \right) + \frac{\partial \ell}{\partial \sigma_\mathcal{B}^2} \cdot \frac{\sum_{i=1}^m -2(x_i - \mu_\mathcal{B})}{m}$$

$$\boxed{\frac{\partial \ell}{\partial x_i} = \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{1}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}} + \frac{\partial \ell}{\partial \sigma_\mathcal{B}^2} \cdot \frac{2(x_i - \mu_\mathcal{B})}{m} + \frac{\partial \ell}{\partial \mu_\mathcal{B}} \cdot \frac{1}{m}}$$

Gradient to propagate to the input layer

$$\frac{\partial \ell}{\partial \gamma} = \sum_{i=1}^m \frac{\partial \ell}{\partial y_i} \cdot \widehat{x}_i$$

$$\frac{\partial \ell}{\partial \beta} = \sum_{i=1}^m \frac{\partial \ell}{\partial y_i}$$

Batch Normalization Gradients

[Ioffe and Szegedy, 2015]

$$\frac{\partial \ell}{\partial \widehat{x}_i} = \frac{\partial \ell}{\partial y_i} \cdot \gamma$$

$$\frac{\partial \ell}{\partial \sigma_\mathcal{B}^2} = \sum_{i=1}^m \frac{\partial \ell}{\partial \widehat{x}_i} \cdot (x_i - \mu_\mathcal{B}) \cdot \frac{-1}{2}(\sigma_\mathcal{B}^2 + \epsilon)^{-3/2}$$

$$\frac{\partial \ell}{\partial \mu_\mathcal{B}} = \left(\sum_{i=1}^m \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_\mathcal{B}^2+\epsilon}}\right) + \frac{\partial \ell}{\partial \sigma_\mathcal{B}^2} \cdot \frac{\sum_{i=1}^m -2(x_i-\mu_\mathcal{B})}{m}$$

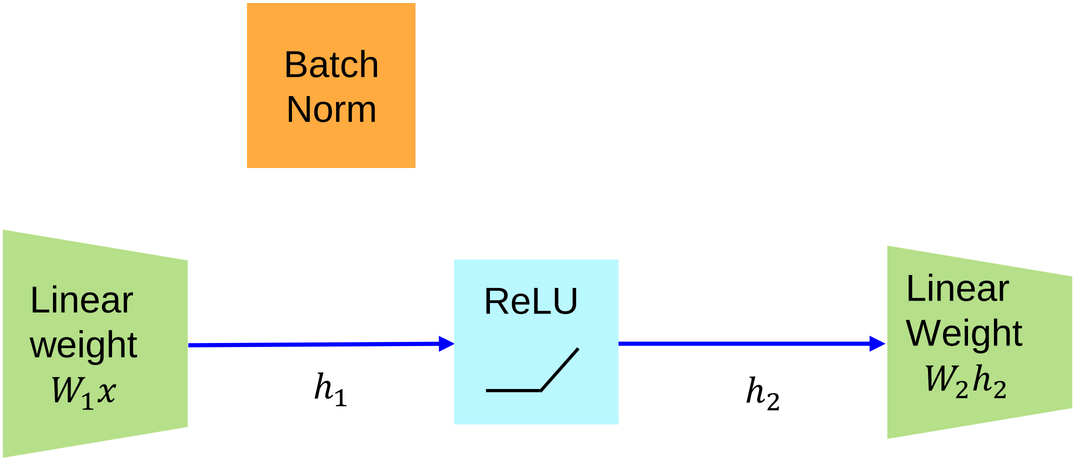
$$\frac{\partial \ell}{\partial x_i} = \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{1}{\sqrt{\sigma_\mathcal{B}^2+\epsilon}} + \frac{\partial \ell}{\partial \sigma_\mathcal{B}^2} \cdot \frac{2(x_i-\mu_\mathcal{B})}{m} + \frac{\partial \ell}{\partial \mu_\mathcal{B}} \cdot \frac{1}{m}$$

$$\boxed{\begin{aligned}\frac{\partial \ell}{\partial \gamma} &= \sum_{i=1}^m \frac{\partial \ell}{\partial y_i} \cdot \widehat{x}_i \\ \frac{\partial \ell}{\partial \beta} &= \sum_{i=1}^m \frac{\partial \ell}{\partial y_i}\end{aligned}}$$

Gradients for the learnable parameters γ and β.

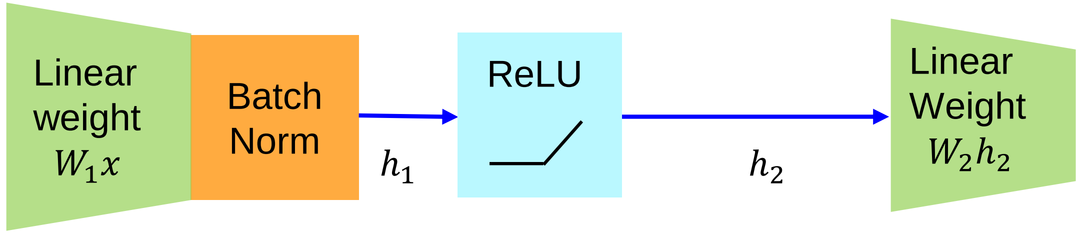
Where to do BatchNorm ?

BatchNorm is just a linear scale/bias layer in the limit of large batch sizes (μ, σ^2).



Where to do BatchNorm ?

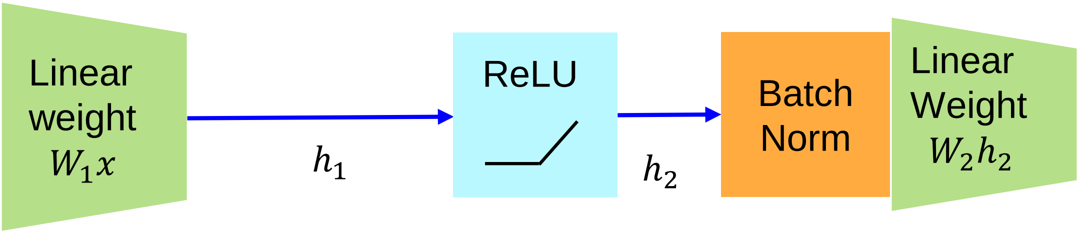
BatchNorm is just a linear scale/bias layer in the limit of large batch sizes (μ, σ^2).



?? Linear weight layer should already have optimal scale/bias in its output

Where to do BatchNorm ?

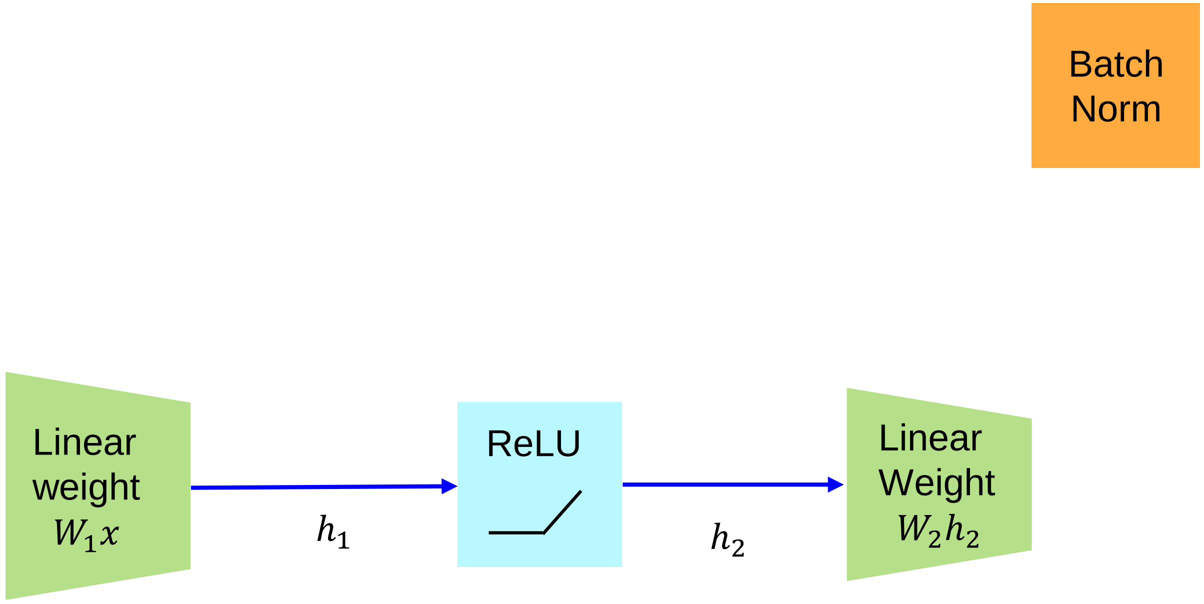
BatchNorm is just a linear scale/bias layer in the limit of large batch sizes (μ, σ^2).



?? Any bias/scaling by the batch norm layer should be over-ruled by the second linear weight layer.

Where to do BatchNorm ?

Best answer: try it in your network. Authors have had different results on different networks.



Why does it work?

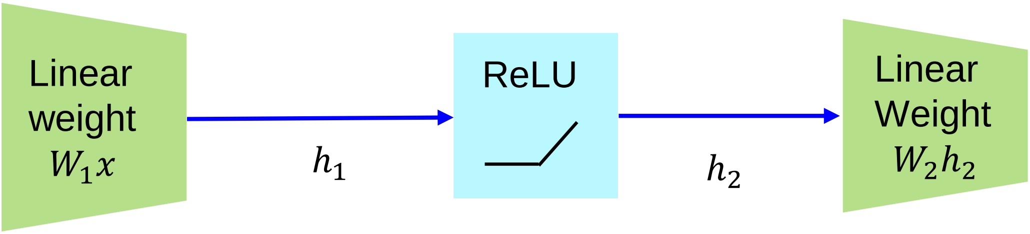
Hard to argue that it is doing normalization, since it will often learn the identity…

Is it really a pseudo-random regularizer (via $1/\sqrt{\sigma^2}$), like dropout?

- Seems to reduce need for dropout

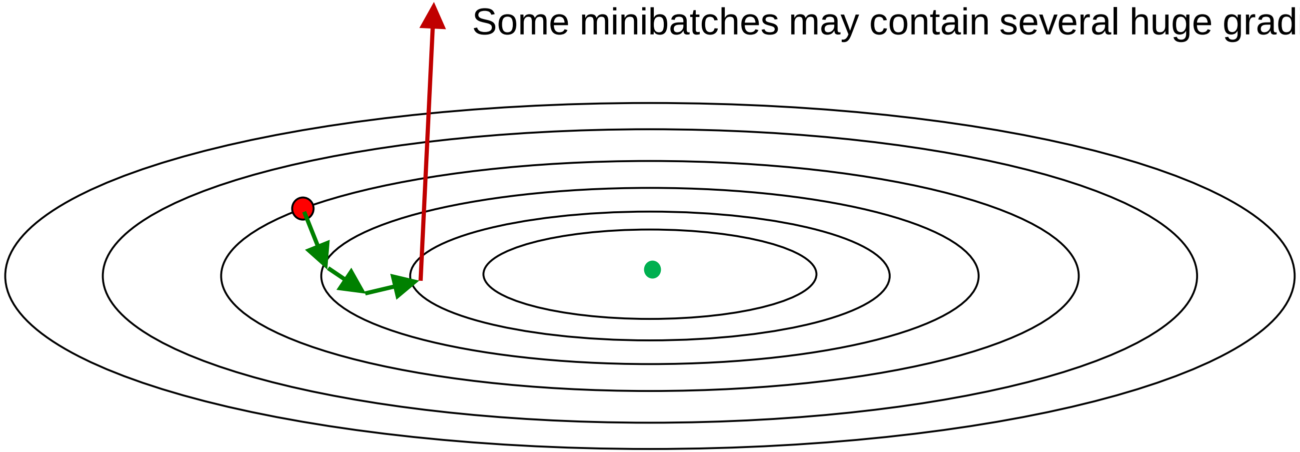
Does it enact a kind of activation/gradient clipping?

- Allows higher learning rates

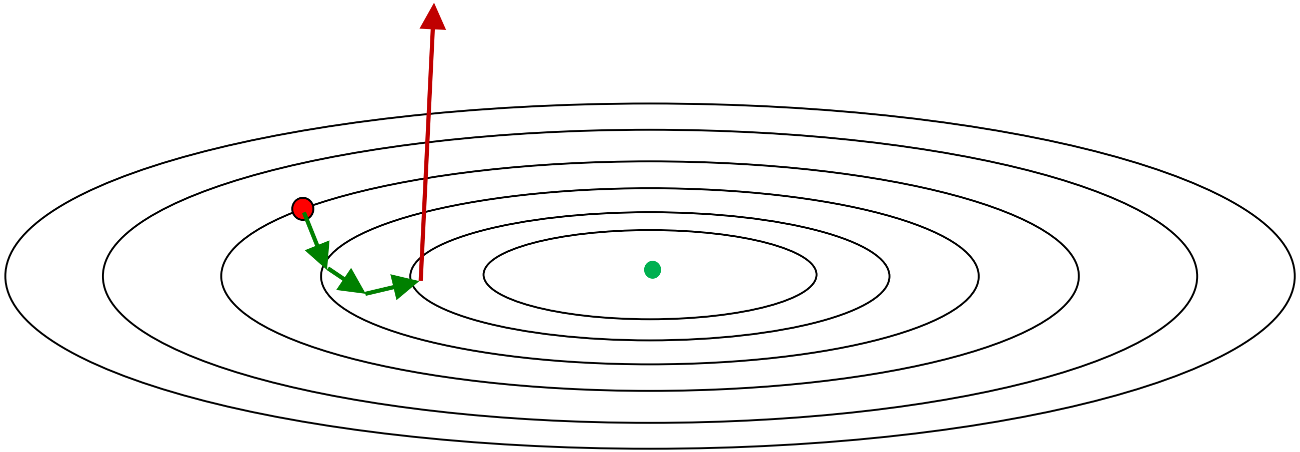


Gradient Magnitudes:

Occasional “monster” gradients
Gradient magnitudes far from gaussian, long-tailed distributions
Gradients much larger for mis-classified instances
Some minibatches may contain several huge gradients



Gradient Clipping by Value:

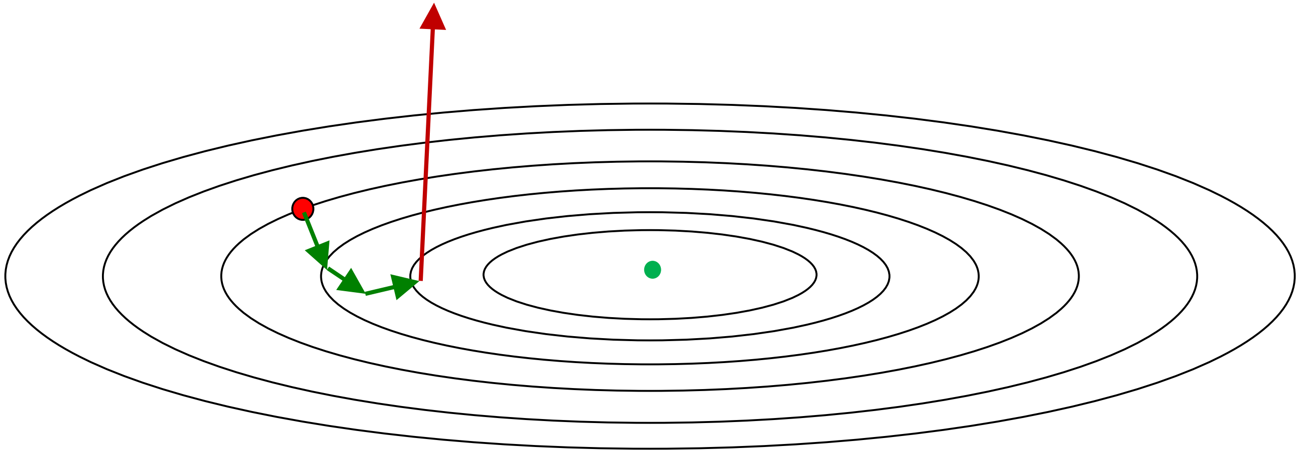


Simply limit the magnitude of each gradient:

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

so $|\bar{g}_i| \leq g_{\max}$. But how to set $g_{\max}$? Use minibatch stats?

Gradient Clipping by Norm:

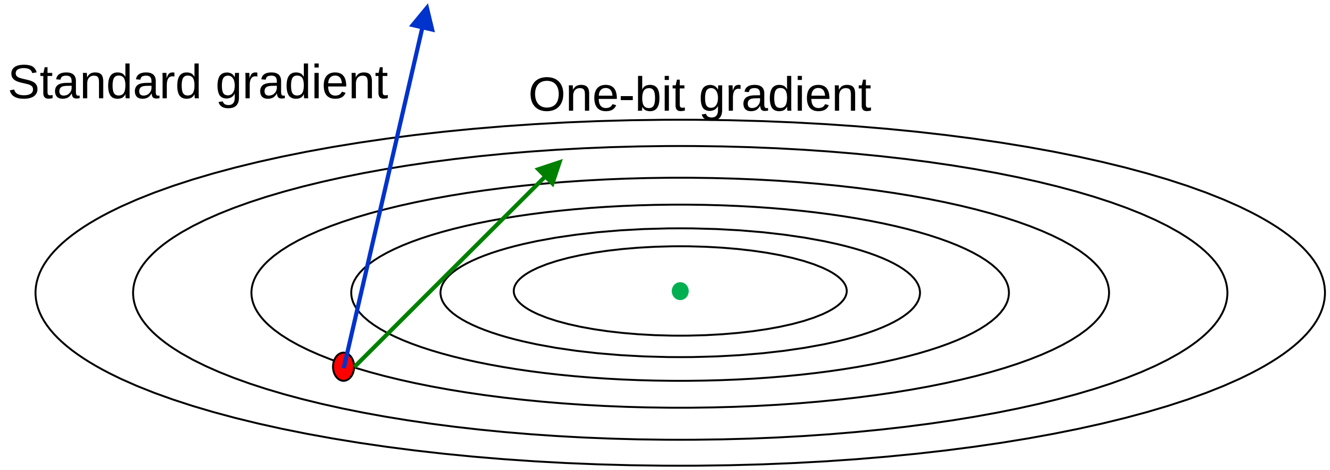


Clip to limit the norm of the gradient:

clipped_grad[i] = grad[i] * clip_norm / max(norm(grad), clip_norm)

Still need to set clip_norm, use a multiple of median norm(grad) ?

One-bit Gradients!



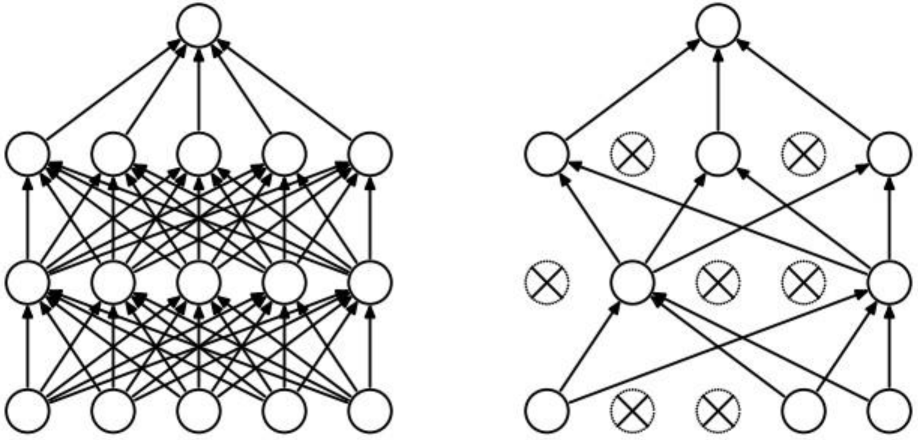
If we clip all gradient dimensions, we are left only with their sign: $\bar{g}_i = g_{\max}(-1,1,1,-1,1,\ldots)$

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)

Dropout

"randomly set some neurons to zero in the forward pass"

i.e. multiply by random bernoulli variables with parameter p.



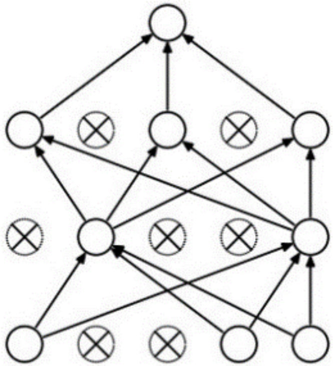
(a) Standard Neural Net (b) After applying dropout.

Note, p is the probability of keeping a neuron (note, incorrect in assignment 1)

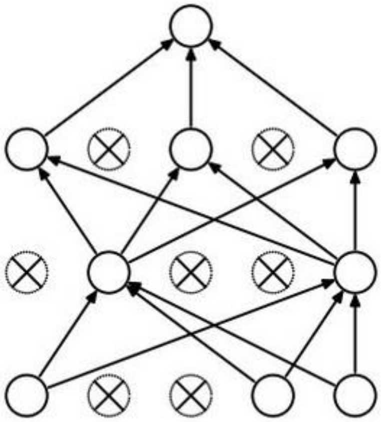
[Srivastava et al., 2014]

```python
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)
```

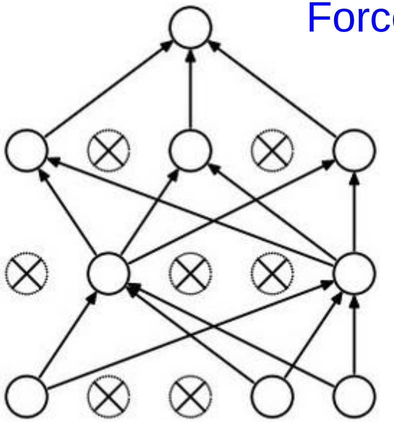
Example forward pass with a 3-layer network using dropout



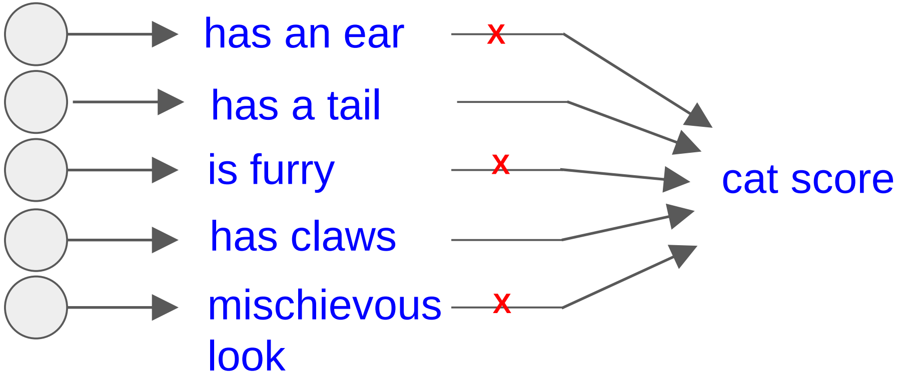
Waaaait a second…
How could this possibly be a good idea?



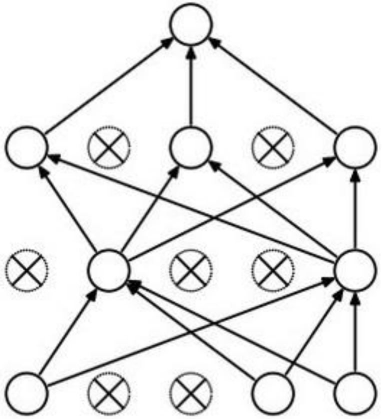
Waaaait a second…
How could this possibly be a good idea?



Forces the network to have a redundant representation.



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



Another interpretation:

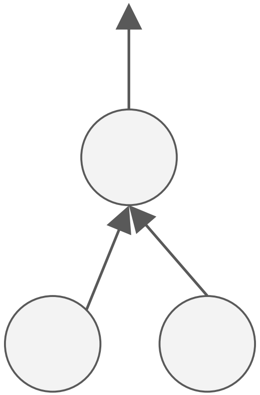
Dropout is training a large ensemble of models (that share parameters).

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

At test time….

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

Leave all input neurons turned on (no dropout).

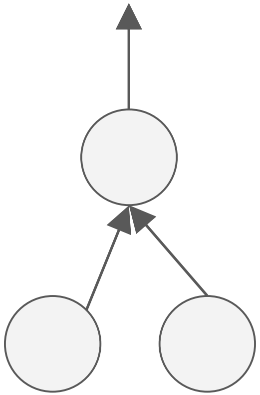


(this can be shown to be an approximation to evaluating the whole ensemble)

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)

We can do something approximate analytically

```python
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"

```python
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!

Ensembles

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.
  Models trained independently.

- 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.
  Models trained independently.

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

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.
  So works best with models that don't have much bias.
  Try to make errors in the models independent – e.g. by training them on different data with bootstrap sampling.

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): Often used with deep learning models.

- Boosting: Rarely used with deep learning models

Why do you think that is?

Ensemble Learning

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

- Bagging: (Bootstrap AGgregation): Often used with deep learning models.
  Models take a long time to train, bagging trivially parallelizes.
  Model differences seem to be mostly due to variance.

- Boosting: Rarely used with deep learning models.
  Deep models are very powerful, not obvious that they have much bias, so unclear that boosting will help.

Ensemble Learning

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

- Bagging: (Bootstrap AGgregation): Often used with deep learning models.

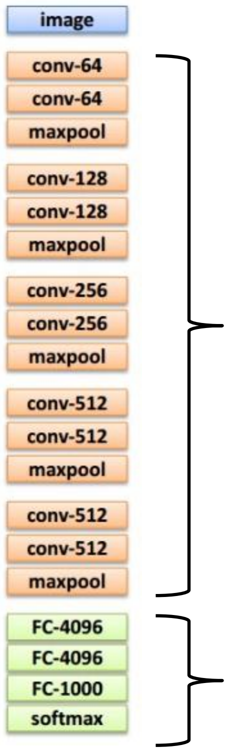
- Boosting: Rarely used with deep learning models.
  Deep networks arguably already correct bias through additional layers.
  This is especially true of Resnets.

Ensemble Approaches

True Ensemble: train several models independently. Combining them:

- Prediction averaging: averaged predicted probabilities, or just vote. Always works.
- Parameter averaging: average the parameters of the models. Almost never works (too many different equivalent parametrizations).

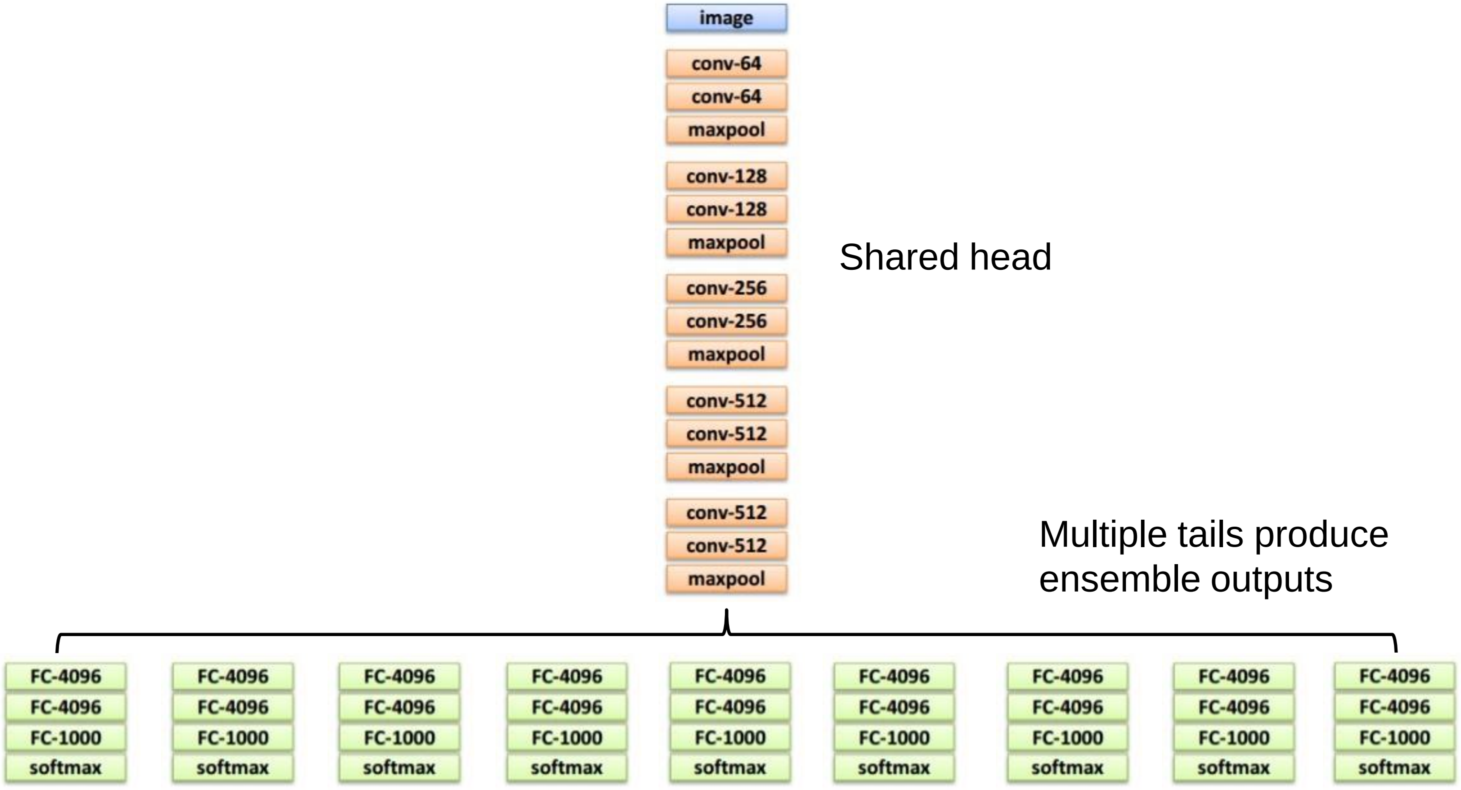
Fast Pseudo-Ensembles



Lots of computational expense, task-independent features

Task and data-dependent

Fast Pseudo-Ensembles



Shared head

Multiple tails produce ensemble outputs

Ensemble Approaches

True Ensemble: train several models independently. Combining them:

- Prediction averaging: averaged predicted probabilities, or just vote. Always works.
- Parameter averaging: average the parameters of the models. Almost never works (too many different equivalent parametrizations).

Model Snapshots: Just keep training a single base model (with fairly high learning rate), and take periodic snapshots of its parameters.

Ensemble Approaches

True Ensemble: train several models independently. Combining them:

- Prediction averaging: Always works.
- Parameter averaging: Almost never works.

Model Snapshots: Just keep training a single base model (with fairly high learning rate), and take periodic snapshots of its parameters.

- Prediction averaging: Always works.
- Parameter averaging: Often works! Snapshots are sufficiently close in parameter space!

- Parameter averaging means you only need to keep a single model for future predictions. If you use a moving average of the snapshots, you only keep one extra set of parameters.

Ensemble Comparisons (VGGNet and CIFAR 10)

Model	Prediction method	Test Accuracy
Baseline (10 epochs)	Single model	0.837
True ensemble of 10 models	Average predictions	0.855
True ensemble of 10 models	Voting	0.851
Snapshots (25) over 10 epochs	Average predictions	0.865
Snapshots (25) over 10 epochs	Voting	0.861
Snapshots (25) over 10 epochs	Parameter averaging	0.864

Ensemble Comparisons (VGGNet and CIFAR 10)

10x compute, 10x storage

Model	Prediction method	Test Accuracy
Baseline (10 epochs)	Single model	0.837
True ensemble of 10 models	Average predictions	0.855
True ensemble of 10 models	Voting	0.851
Snapshots (25) over 10 epochs	Average predictions	0.865
Snapshots (25) over 10 epochs	Voting	0.861
Snapshots (25) over 10 epochs	Parameter averaging	0.864

Ensemble Comparisons (VGGNet and CIFAR 10)

Model	Prediction method	Test Accuracy
Baseline (10 epochs)	Single model	0.837
True ensemble of 10 models	Average predictions	0.855
True ensemble of 10 models	Voting	0.851
Snapshots (25) over 10 epochs	Average predictions	0.865
Snapshots (25) over 10 epochs	Voting	0.861
Snapshots (25) over 10 epochs	Parameter averaging	0.864

1x compute, 25x storage

Ensemble Comparisons (VGGNet and CIFAR 10)

Model	Prediction method	Test Accuracy
Baseline (10 epochs)	Single model	0.837
True ensemble of 10 models	Average predictions	0.855
True ensemble of 10 models	Voting	0.851
Snapshots (25) over 10 epochs	Average predictions	0.865
Snapshots (25) over 10 epochs	Voting	0.861
Snapshots (25) over 10 epochs	Parameter averaging	**0.864**

1x compute, 2x storage

Ensemble Comparisons (VGGNet and CIFAR 10)

Model	Prediction method	Test Accuracy
Baseline (10 epochs)	Single model	0.837
True ensemble of 10 models	Average predictions	0.855
True ensemble of 10 models	Voting	0.851
Snapshots (25) over 10 epochs	Average predictions	0.865
Snapshots (25) over 10 epochs	Voting	0.861
Snapshots (25) over 10 epochs	Parameter averaging	**0.864**

See also:

Distilling the Knowledge in a Neural Network, Geoffrey Hinton, Oriol Vinyals, Jeff Dean, arXiv 1503.02531

Constructs a single (better) model from an ensemble.

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:

Setting	Best Test Accuracy	Average Test Accuracy
Experiment 1: Simple Init, No Gradient Clipping		
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%

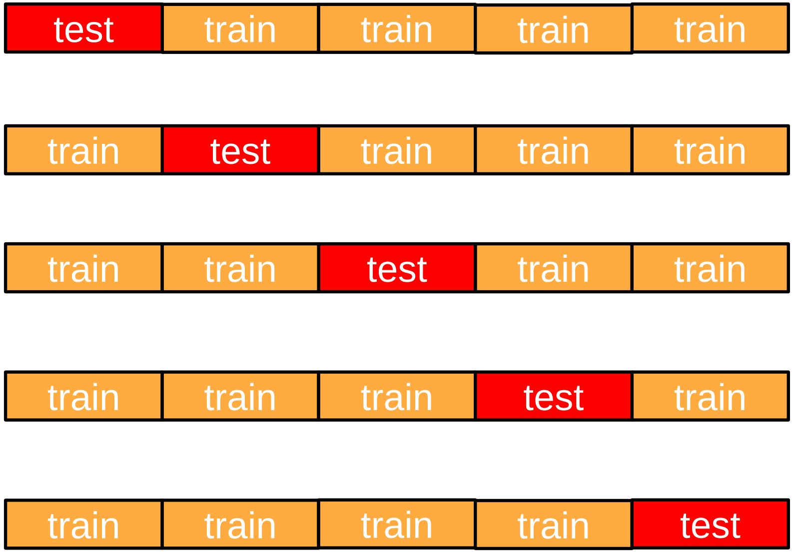
Table 1: Average and best test accuracy percentages on MNIST over 40 runs. Higher values are better.

This Time: Hyperparameter Optimization

Hyperparameters: learning rate, momentum decay, dropout rate,…

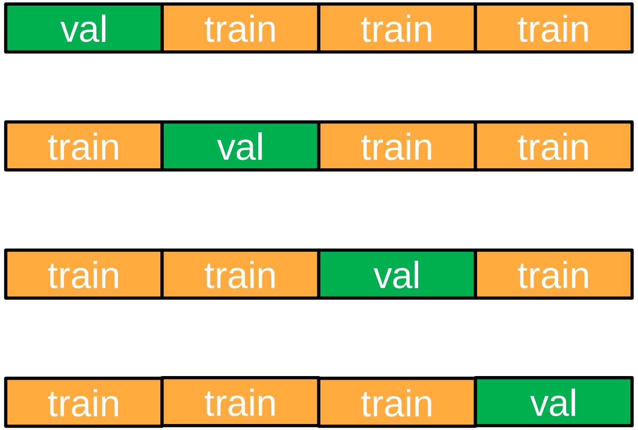
Hyperparameter Optimization

Normal Cross-Validation: Use a blocked design for testing



Hyperparameter Optimization

Cross-Validation: You can use a validation set(s) for hyperparameter evaluation/tuning ***within each training block***.



Cross-validation strategy

coarse -> fine cross-validation in stages

First stage: only a few epochs, wide range of parameter values to get rough idea of what params work
Second stage: longer running time, finer search
… (repeat as necessary)

Tip for detecting explosions in the solver:
If the cost is ever > 3 * original cost, break out early

For example: run coarse search for 5 epochs

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-5, 5)
      lr = 10**uniform(-3, -6)

      trainer = ClassifierTrainer()
      model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
      trainer = ClassifierTrainer()
      best_model_local, stats = trainer.train(X_train, y_train, X_val, y_val,
                                   model, two_layer_net,
                                   num_epochs=5, reg=reg,
                                   update='momentum', learning_rate_decay=0.9,
                                   sample_batches = True, batch_size = 100,
                                   learning_rate=lr, verbose=False)
```

note it's best to optimize in log space!

```
val_acc: 0.412000, lr: 1.405206e-04, reg: 4.793564e-01, (1 / 100)
val_acc: 0.214000, lr: 7.231888e-06, reg: 2.321281e-04, (2 / 100)
val_acc: 0.208000, lr: 2.119571e-06, reg: 8.011857e+01, (3 / 100)
val_acc: 0.196000, lr: 1.551131e-05, reg: 4.374936e-05, (4 / 100)
val_acc: 0.079000, lr: 1.753300e-05, reg: 1.200424e+03, (5 / 100)
val_acc: 0.223000, lr: 4.215128e-05, reg: 4.196174e+01, (6 / 100)
val_acc: 0.441000, lr: 1.750259e-04, reg: 2.110807e-04, (7 / 100)
val_acc: 0.241000, lr: 6.749231e-05, reg: 4.226413e+01, (8 / 100)
val_acc: 0.482000, lr: 4.296863e-04, reg: 6.642555e-01, (9 / 100)
val_acc: 0.079000, lr: 5.401602e-06, reg: 1.599828e+04, (10 / 100)
val_acc: 0.154000, lr: 1.618508e-06, reg: 4.925252e-01, (11 / 100)
```

nice

Now run finer search...

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-5, 5)
      lr = 10**uniform(-3, -6)
```

adjust range

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-4, 0)
      lr = 10**uniform(-3, -4)
```

```
val_acc: 0.527000, lr: 5.340517e-04, reg: 4.097824e-01, (0 / 100)
val_acc: 0.492000, lr: 2.279484e-04, reg: 9.991345e-04, (1 / 100)
val_acc: 0.512000, lr: 8.680827e-04, reg: 1.349727e-02, (2 / 100)
val_acc: 0.461000, lr: 1.028377e-04, reg: 1.220193e-02, (3 / 100)
val_acc: 0.460000, lr: 1.113730e-04, reg: 5.244309e-02, (4 / 100)
val_acc: 0.498000, lr: 9.477776e-04, reg: 2.001293e-03, (5 / 100)
val_acc: 0.469000, lr: 1.484369e-04, reg: 4.328313e-01, (6 / 100)
val_acc: 0.522000, lr: 5.586261e-04, reg: 2.312685e-04, (7 / 100)
val_acc: 0.530000, lr: 5.808183e-04, reg: 8.259964e-02, (8 / 100)
val_acc: 0.489000, lr: 1.979168e-04, reg: 1.010889e-04, (9 / 100)
val_acc: 0.490000, lr: 2.036031e-04, reg: 2.406271e-03, (10 / 100)
val_acc: 0.475000, lr: 2.021162e-04, reg: 2.287807e-01, (11 / 100)
val_acc: 0.460000, lr: 1.135527e-04, reg: 3.905040e-02, (12 / 100)
val_acc: 0.515000, lr: 6.947668e-04, reg: 1.562808e-02, (13 / 100)
val_acc: 0.531000, lr: 9.471549e-04, reg: 1.433895e-03, (14 / 100)
val_acc: 0.509000, lr: 3.140888e-04, reg: 2.857518e-01, (15 / 100)
val_acc: 0.514000, lr: 6.438349e-04, reg: 3.033781e-01, (16 / 100)
val_acc: 0.502000, lr: 3.921784e-04, reg: 2.707126e-04, (17 / 100)
val_acc: 0.509000, lr: 9.752279e-04, reg: 2.850865e-03, (18 / 100)
val_acc: 0.500000, lr: 2.412048e-04, reg: 4.997821e-04, (19 / 100)
val_acc: 0.466000, lr: 1.319314e-04, reg: 1.189915e-02, (20 / 100)
val_acc: 0.516000, lr: 8.039527e-04, reg: 1.528291e-02, (21 / 100)
```

53% - relatively good for a 2-layer neural net with 50 hidden neurons.

Now run finer search...

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-5, 5)
      lr = 10**uniform(-3, -6)
```

adjust range

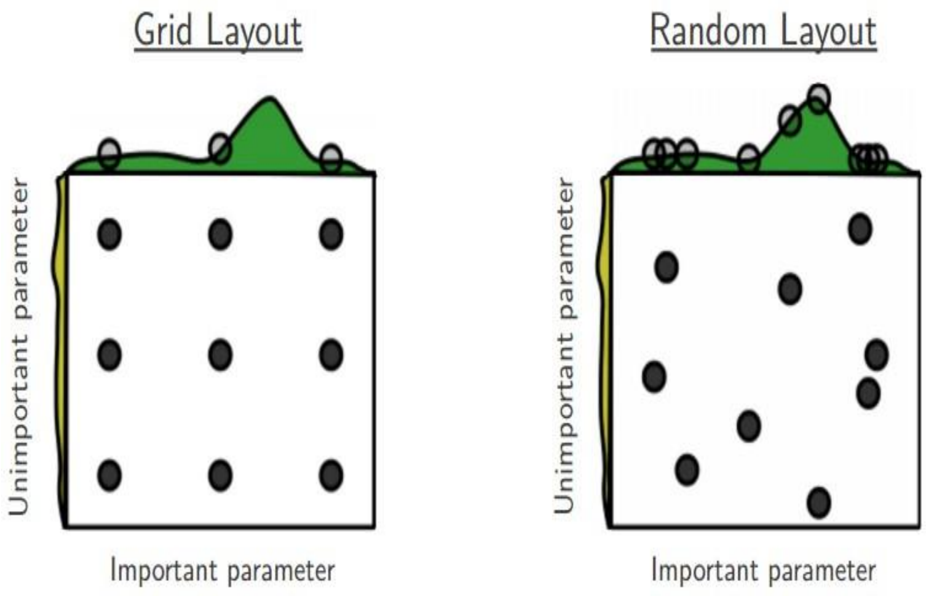
```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-4, 0)
      lr = 10**uniform(-3, -4)
```

```
val_acc: 0.527000, lr: 5.340517e-04, reg: 4.097824e-01, (0 / 100)
val_acc: 0.492000, lr: 2.279484e-04, reg: 9.991345e-04, (1 / 100)
val_acc: 0.512000, lr: 8.680827e-04, reg: 1.349727e-02, (2 / 100)
val_acc: 0.461000, lr: 1.028377e-04, reg: 1.220193e-02, (3 / 100)
val_acc: 0.460000, lr: 1.113730e-04, reg: 5.244309e-02, (4 / 100)
val_acc: 0.498000, lr: 9.477776e-04, reg: 2.001293e-03, (5 / 100)
val_acc: 0.469000, lr: 1.484369e-04, reg: 4.328313e-01, (6 / 100)
val_acc: 0.522000, lr: 5.586261e-04, reg: 2.312685e-04, (7 / 100)
val_acc: 0.530000, lr: 5.808183e-04, reg: 8.259964e-02, (8 / 100)
val_acc: 0.489000, lr: 1.979168e-04, reg: 1.010889e-04, (9 / 100)
val_acc: 0.490000, lr: 2.036031e-04, reg: 2.406271e-03, (10 / 100)
val_acc: 0.475000, lr: 2.021162e-04, reg: 2.287807e-01, (11 / 100)
val_acc: 0.460000, lr: 1.135527e-04, reg: 3.905040e-02, (12 / 100)
val_acc: 0.515000, lr: 6.947668e-04, reg: 1.562808e-02, (13 / 100)
val_acc: 0.531000, lr: 9.471549e-04, reg: 1.433895e-03, (14 / 100)
val_acc: 0.509000, lr: 3.140888e-04, reg: 2.857518e-01, (15 / 100)
val_acc: 0.514000, lr: 6.438349e-04, reg: 3.033781e-01, (16 / 100)
val_acc: 0.502000, lr: 3.921784e-04, reg: 2.707126e-04, (17 / 100)
val_acc: 0.509000, lr: 9.752279e-04, reg: 2.850865e-03, (18 / 100)
val_acc: 0.500000, lr: 2.412048e-04, reg: 4.997821e-04, (19 / 100)
val_acc: 0.466000, lr: 1.319314e-04, reg: 1.189915e-02, (20 / 100)
val_acc: 0.516000, lr: 8.039527e-04, reg: 1.528291e-02, (21 / 100)
```

53% - relatively good for a 2-layer neural net with 50 hidden neurons.

But this best cross-validation result is worrying. Why?

Random Search vs. Grid Search



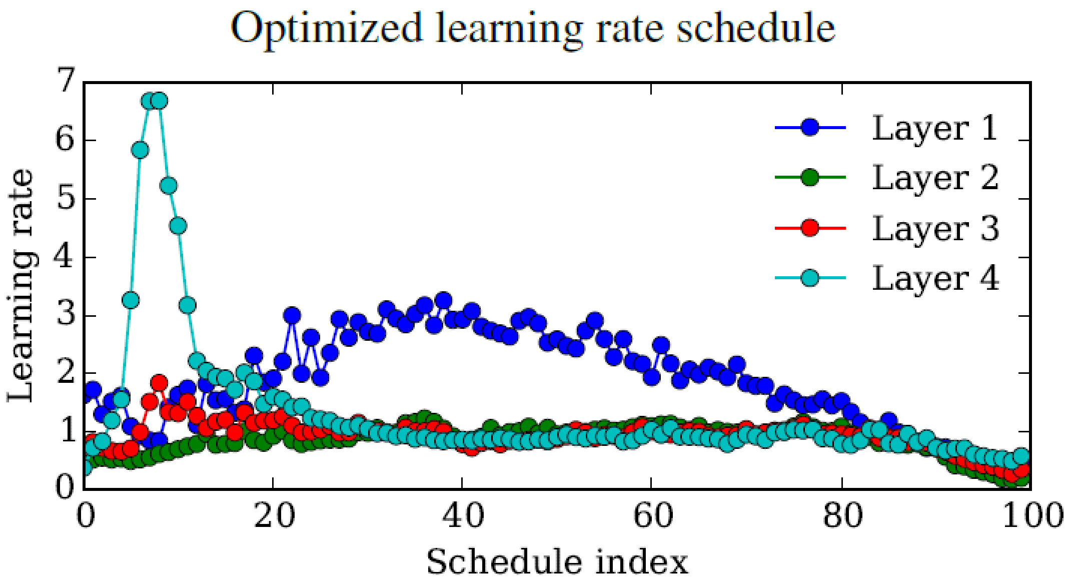
Random Search for Hyper-Parameter Optimization
Bergstra and Bengio, 2012

Hyperparameters to play with:

- network architecture
- learning rate, its decay schedule, update type
- regularization (L2/Dropout strength)

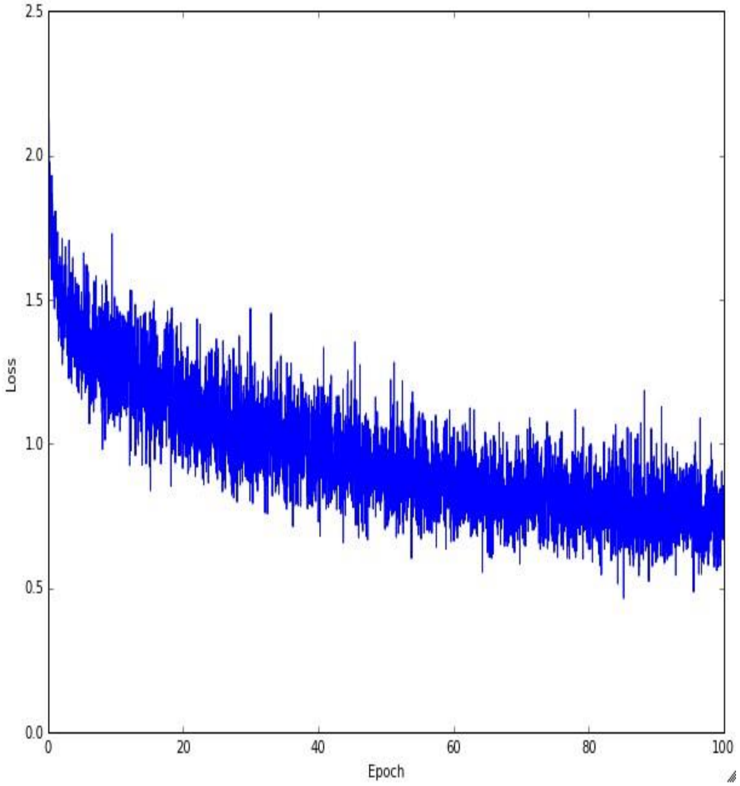
neural networks practitioner
music = loss function

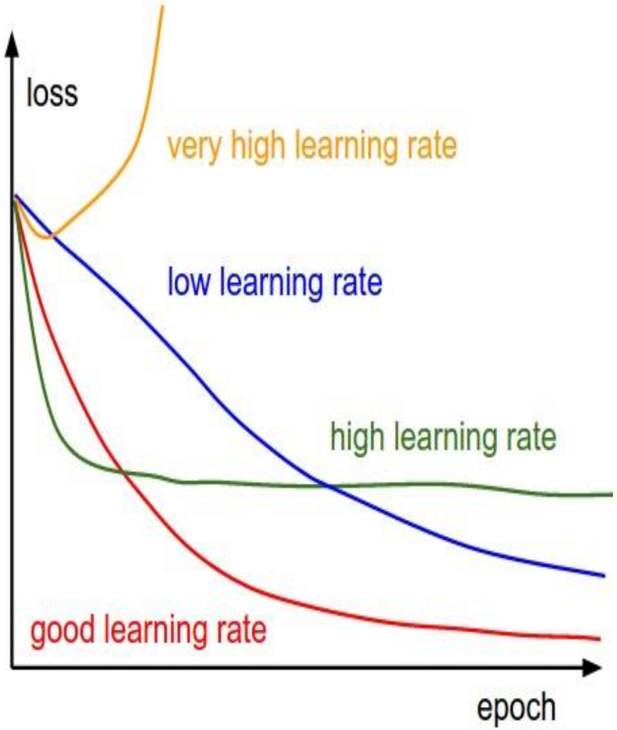
Optimal Hyper-parameters

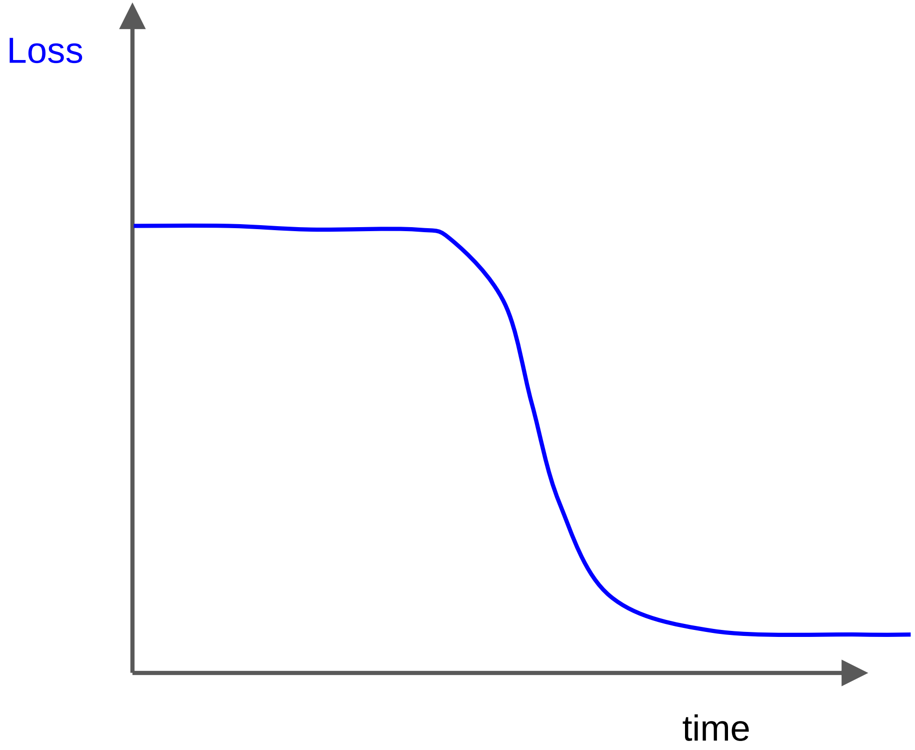


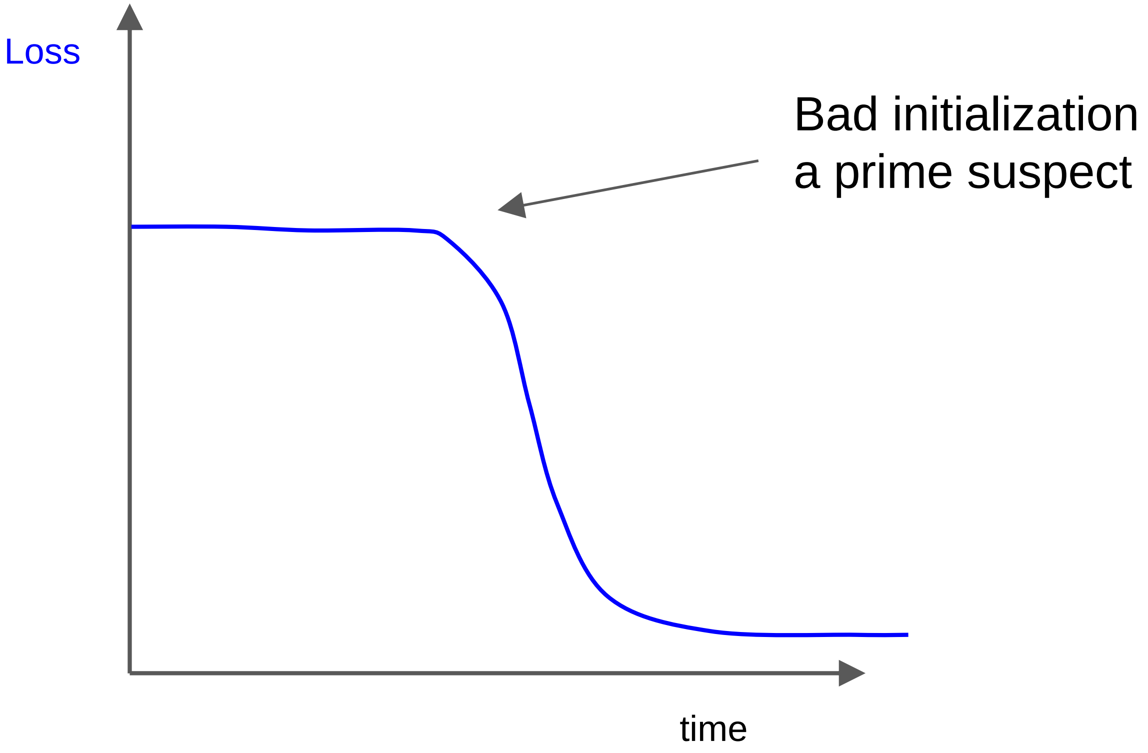
From [“Gradient-based Hyperparameter Optimization through Reversible Learning”] Dougal et al., 2015

Monitor and visualize the loss curve

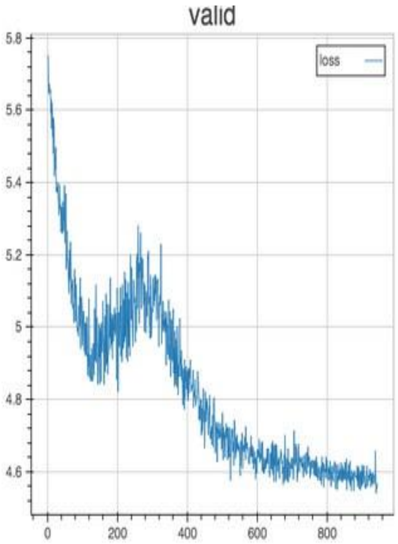


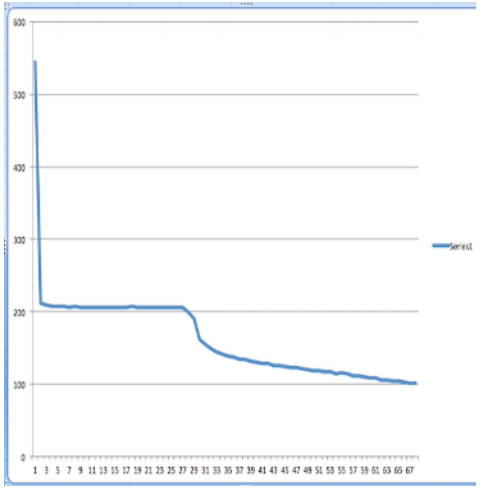


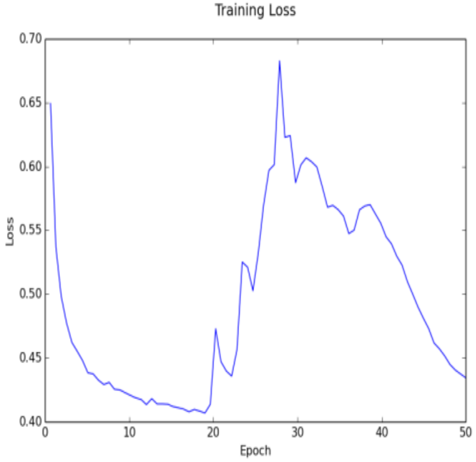

Loss
time


Loss
Bad initialization
a prime suspect
time

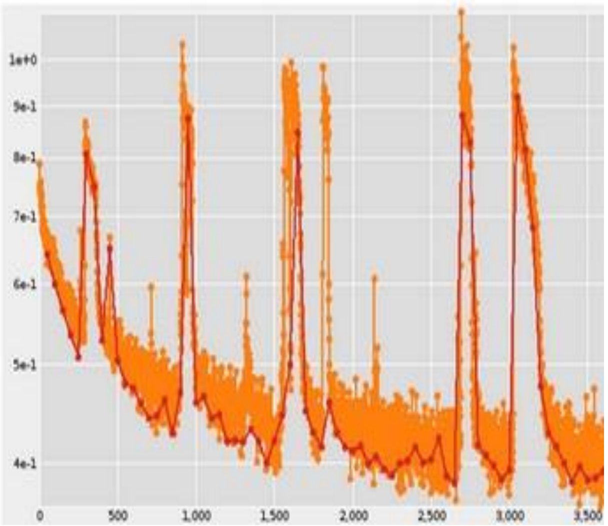
lossfunctions.tumblr.com

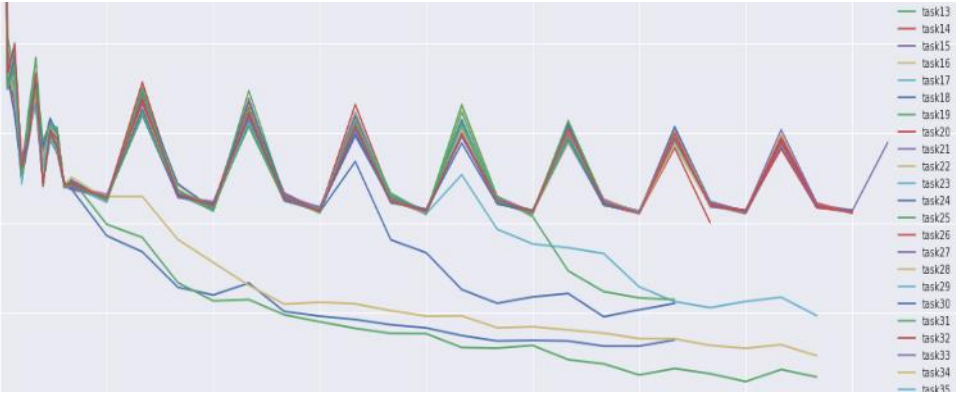


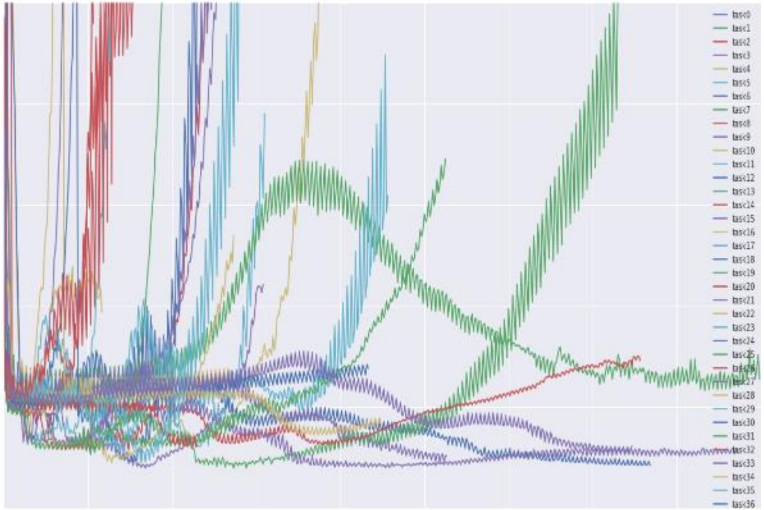




lossfunctions.tumblr.com





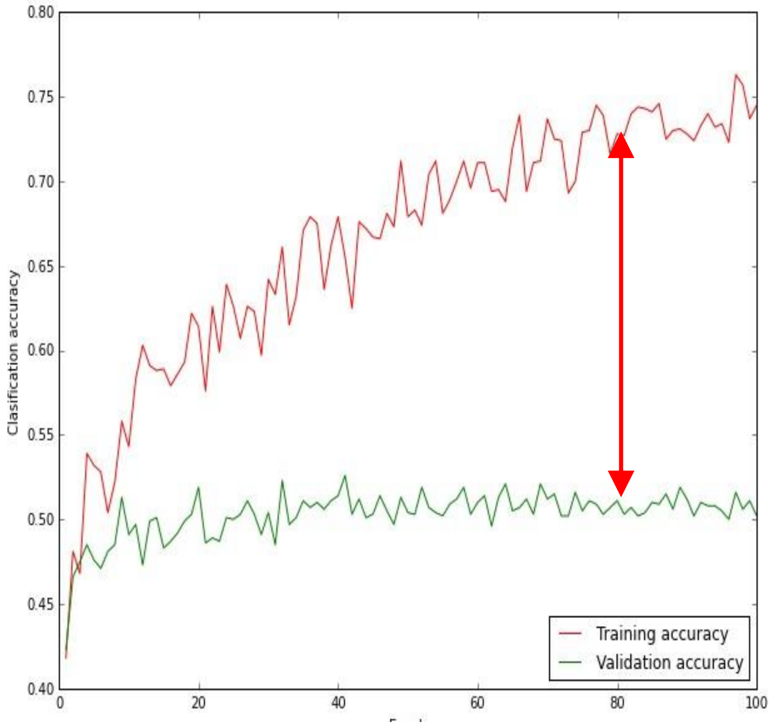


lossfunctions.tumblr.com

Other Features to capture and plot

- Per-layer activations:
  - Magnitude, center (mean or median), breadth (sdev or quartiles)
  - Spatial/feature-rank variations
- Gradients
  - Magnitude, center (mean or median), breadth (sdev or quartiles)
  - Spatial/feature-rank variations
- Learning trajectories
  - Plot parameter values in a low-dimensional space

Monitor and visualize the accuracy:



big gap = overfitting

=> increase regularization strength?

no gap

=> increase model capacity?

Track the ratio of weight updates / weight magnitudes:

```
# assume parameter vector W and its gradient vector dW
param_scale = np.linalg.norm(W.ravel())
update = -learning_rate*dW # simple SGD update
update_scale = np.linalg.norm(update.ravel())
W += update # the actual update
print update_scale / param_scale # want ~1e-3
```

ratio between the values and updates: ~ 0.0002 / 0.02 = 0.01 (about okay)
want this to be somewhere around 0.001 or so

Summary

- Batch Normalization
- Gradient clipping & one-bit gradients
- Dropout
- Ensembles
- Gradient noise
- Hyperparameter optimization