# OS4118
# Neural Networks Part III

https://devblogs.nvidia.com/parallelforall/deep-learning-self-driving-cars/

# Let’s Keep Talking

- We’ve seen that we can build pretty accurate NN models…
- But also bad ones, and we don’t have much guidance on how to select parameters (numbers and sizes of layers, regularization, and so on)
- We need to be smarter about training. Just regularizing wont do it. We will talk about a few “old” approaches; learn a bit about **gradient descent**; think about the error surface what we might do to make gradient descent work better…
- …and then talk about a couple of new approaches to make training a **Deep NN** feasible.

# A Little Bit About Visualizing Features of a NN

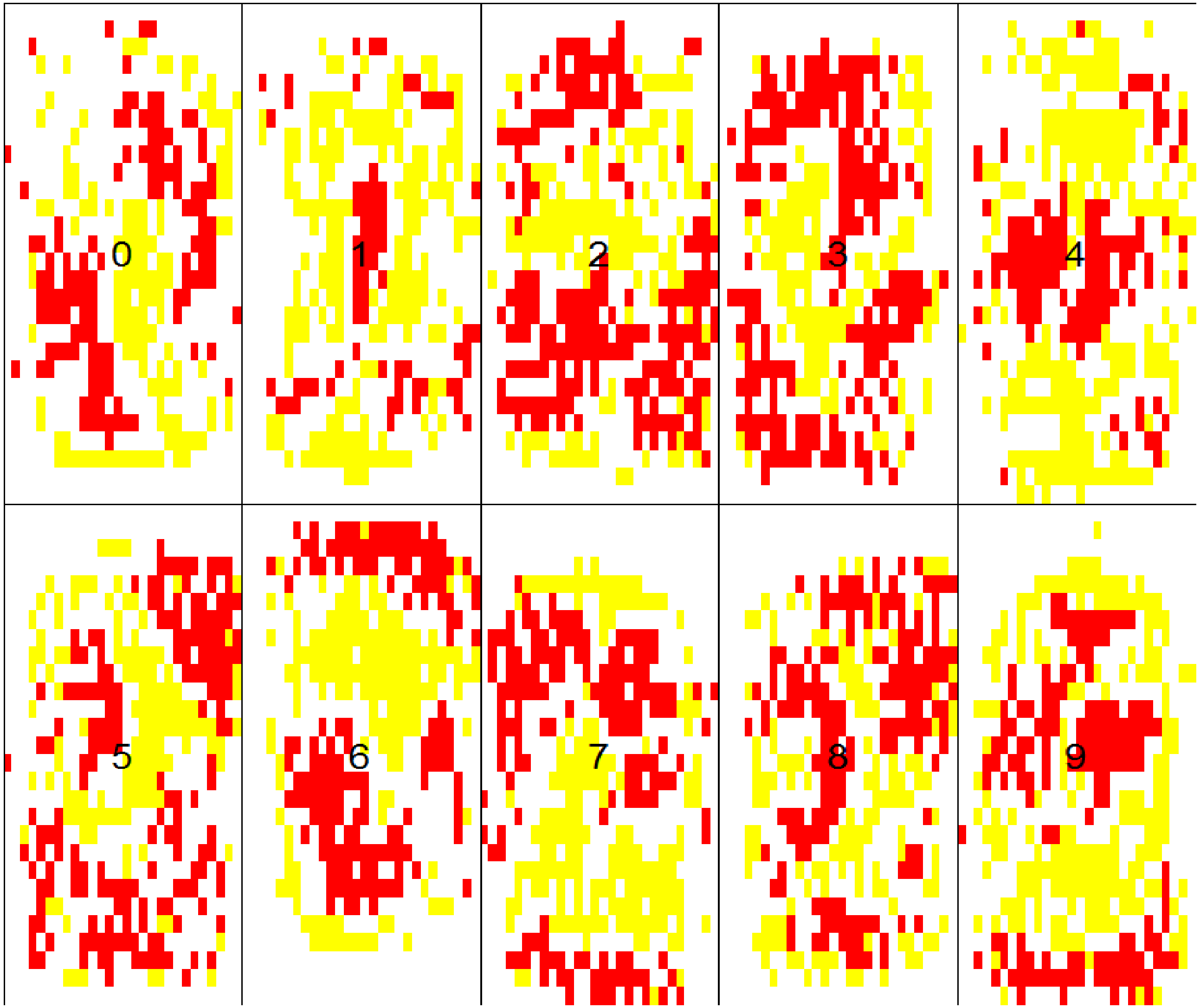
# Visualizing a node output

## Or an image of the input that most strongly activates a node.

A simple hand - written digit recognizer based on MNIST : 28 x 28 + 1 = 785 input nodes, 10 output nodes, no hidden layer.

Here we see which types of pixel values most strongly activate each output node *( increase the output value or decrease the value).*

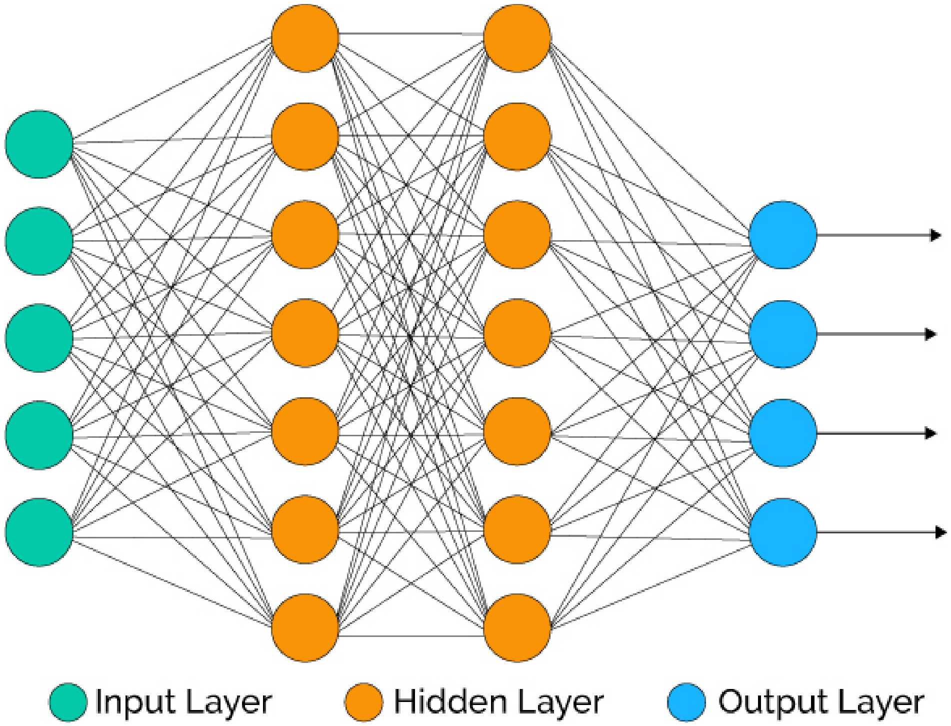
Red = high pixel values here increase the output
Yellow = high pixel values decrease the output
White = Doesn't matter.



# Visualizing a node output

Which values of the input activate a node the most?

For the first node, the answer is easy.



# What values of the *x*'s maximize the output of the second unit in the first hidden layer?

## What $x$'s maximize $u_2$?

Since the activation function, f, is non-decreasing, $u_2$ is maximized by maximizing the unit input

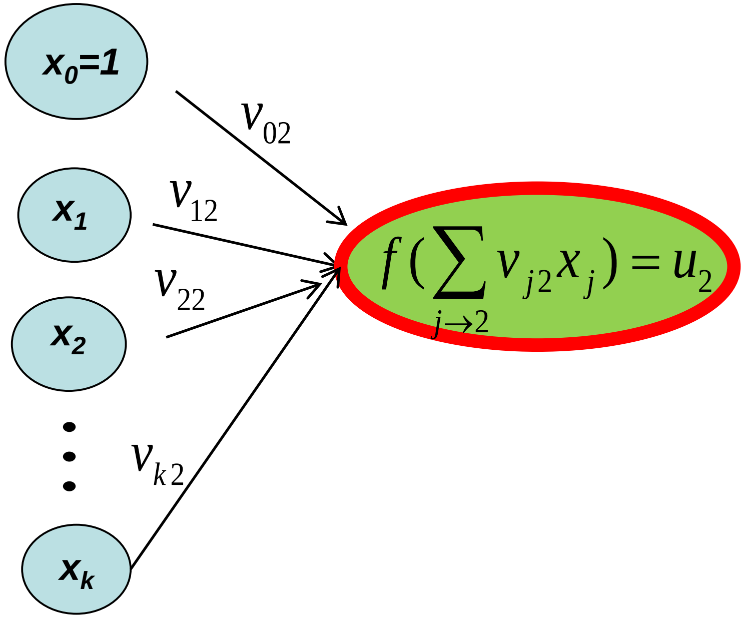
$$z = \sum_{i=1}^{k} v_{i2} x_{i2} + v_{02}(1)$$

WLOG, restrict attention to inputs constrained to have unit length

i.e. $\sum_{i=1}^{n} x_i^2 = 1.$

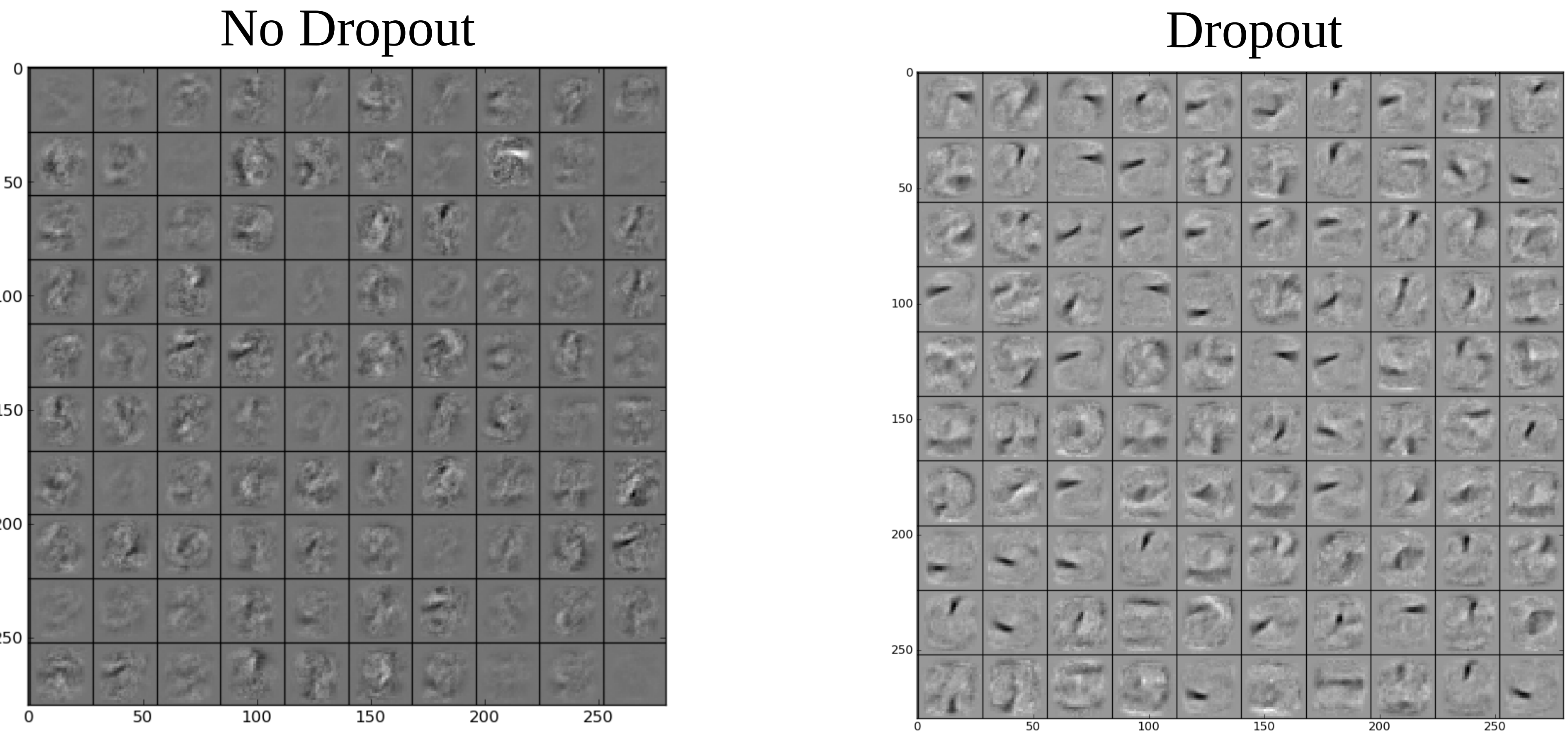
The unit input is maximized by

$$x_i = \frac{v_{i2}}{\sum_j (v_{j2})^2} \qquad \text{for } i = 1, \ldots, k.$$



## $x$'s proportional to their weights

# Visualizing the output of  nodes in the first hidden layer show what features are being generated in that layer (Hinton, 2012)

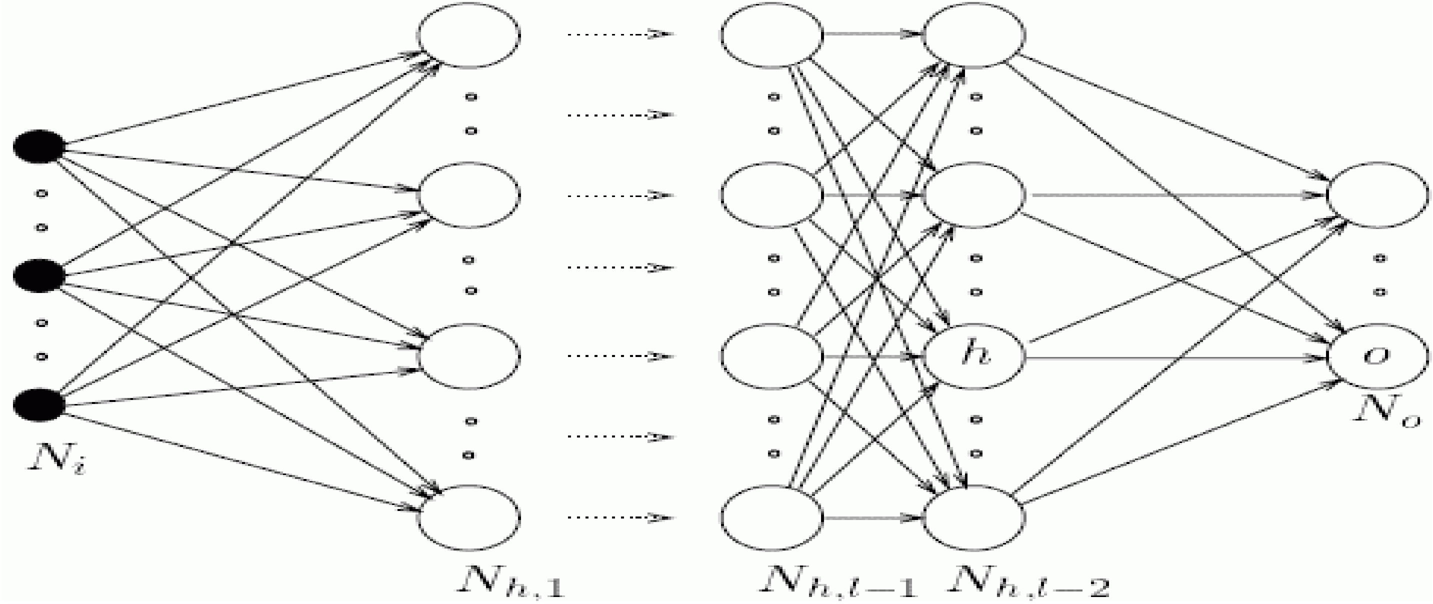


100 “learned” features taken from the first layer of a 784 – 500 – 500  – 10  network. Each picture shows 28x28 image that maximally activates a node in the first hidden layer (MNIST data)

# Deep NN

## What Next?Add layers….

## ( our automatic feature generators)



http://markus.com/deep-learning-101/

# But....

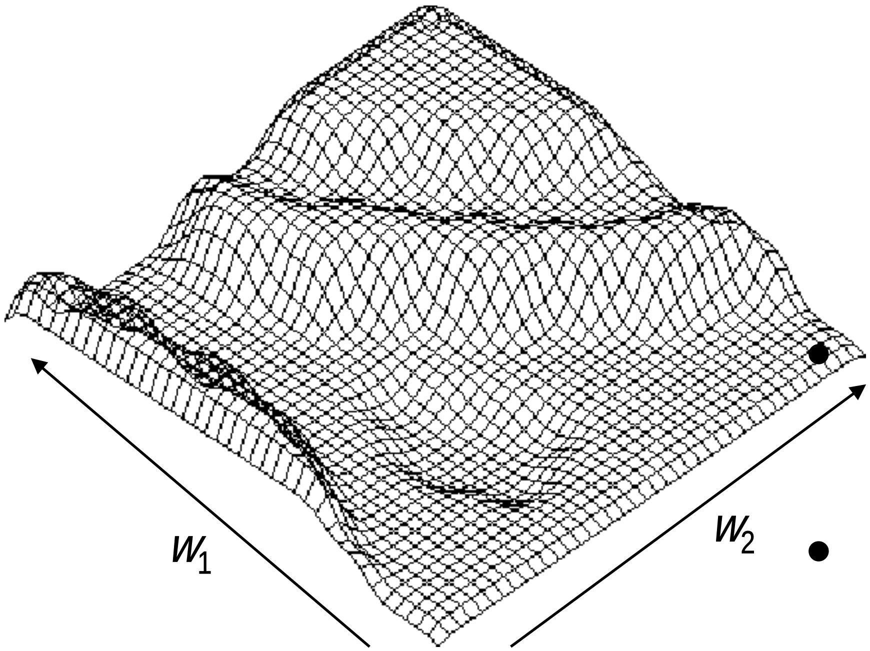
- Nvidia (self-driving car) input is 3 x 30 x 60 image
- Deep NN has 27 million connections and 250 thousand weights (this is not a fully connected NN – how do they do that?)
- We need more data: Tesla (2016) has 1.3 billion miles of data, 8 cameras, one radar. https://www.bloomberg.com/news/articles/2016-12-20/the-tesla-advantage-1-3-billion-miles-of-data
- We need to be smarter about training.

# Training A (Deep) NN

# Step 1

- Set aside a test set. MNIST has a test set of 10,000 images
- For deep NN, you may not have the time, computational power, etc. for cross-validation. Set aside a **validation set** out of the training set to use as a pseudo-test set rather than cross-validating.
- For MNIST, set aside a10,000 validation set out of the training set. That leaves 50,000 for training.
- But, now, we are talking about training **one NN only** with a fixed architecture. We also need to think about choosing the **correct** architecture, trying architectures of different complexity.
- If you only have time for a few, start with a big one and then then regularize the heck out of it.

# Error Surface



- The “error surface” shows how well we approximate ***x***, as a function of the weights
- We want to find the ***w*** that give us minimum error
- During training we start at some ***w***, compute the **gradient**, move downhill, re-compute, etc.

https://www.aanda.org/articles/aa/full/2004/32/aa3658/aa3658.right.html

# Gradient Descent, at the mth step:

- Of course in real life this “surface” might be 100,000-dimensional
- With one pass through the data (an **epoch**) we can compute the gradient exactly
- $\boldsymbol{w}^{m+1} \leftarrow \boldsymbol{w}^{m} - \alpha \left.\frac{\partial E}{\partial \boldsymbol{w}}\right|_{\boldsymbol{w}=\boldsymbol{w}^{m}}$

Gradient here

New weight vector

Old weight vector

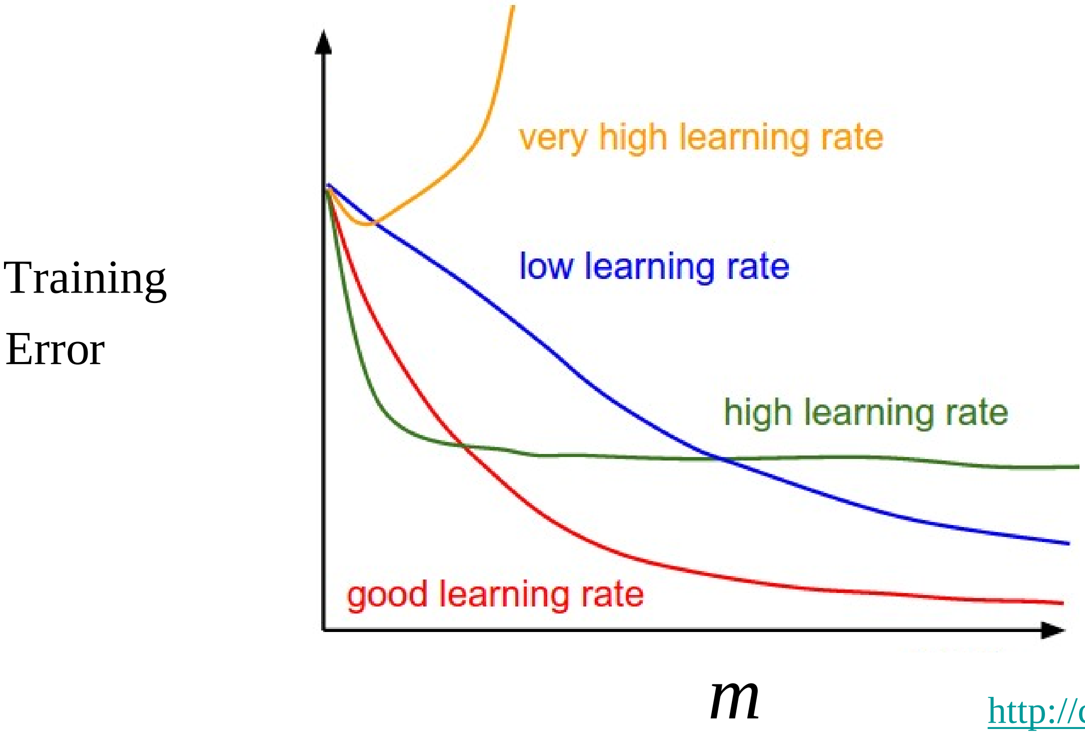
Step size

# Gradient Descent, at the mth step:

- Of course in real life this “surface” might be 100,000-dimensional
- With one pass through the data (an **epoch**) we can compute the gradient exactly
- $\boldsymbol{w}^{m+1} \leftarrow \boldsymbol{w}^{m} - \alpha \left.\frac{\partial E}{\partial \boldsymbol{w}}\right|_{\boldsymbol{w}=\boldsymbol{w}^{m}}$
- That is, take a step of size $\alpha$ in the direction of steepest descent
- We need to choose $\alpha$, while avoiding local minima, flat spots, long narrow valleys, and going either too fast or too slowly

# Going Too Fast? Too Slow?

$$w^{m+1} = w^m - \alpha\, \partial E / \partial w^m$$



http://cs231n.github.io/neural-networks-3/

- Start with a **few** epochs and a **small regularization parameter**. Choose a reasonable learning rate
- Usually works best if, alpha, the step size or **learning rate**, is decreased with training.

# Strategy (ala cs231n): Regularization and Learning Rate

- Now start a grid search on the combination of **regularization** and **learning rate** and **model architecture**. Start coarse with only a few epochs (here look at validation error).

- **Second stage** of grid search, make a finer grid and more epochs.

- Note, **randomizing** the choice of hyper-parameters is better than a grid. Make them Uniform on a **log-scale**

- **Key lesson:** Selecting this parameters is hard

# Avoiding Flat Spots

- A flat spot in the error function for a particular weight, is a region where the gradient for the weight is zero.
- If the |input| to a hidden node is very large, and the activation function has flat spots (e.g. logistic) then the output of the node (the activation) is always 1 ( or always 0). This is node is **"saturated"** or **"killed".**
- Gradients of weights going into a saturated node are zero.
- Fixes (these are good for other reasons too):
  - Control magnitudes of inputs: standardizing them
  - Control magnitudes of weights: regularize ($L_2$ –weight decay or $L_1$) ; small initial random values of weights.
  - Change the activation function: ReLu.

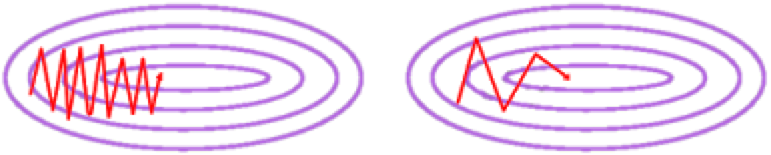
# How about those long valleys?

What causes valleys?

- The Inputs to the node are multi-collinear
- In regression, this means the features (or linear combinations of them) are redundant.
- In a NN, this means that outputs of nodes in the same layer (or linear combinations of them) are redundant. I.e. the layer is defining multi-collinear features, which will cause problems for the nodes in the next layer.

Solutions?

- Slow the change in the direction of descent with “momentum”.



http://ecee.colorado.edu/~ecen4831/lectures/NNet3.html

# Momentum

- Adding **momentum** slows the change in the direction of descent :

$$\mathbf{w}^{m+1}=\mathbf{w}^{m} - \alpha\ [(1-\eta)\ \partial E/\partial \mathbf{w}^{m} + \eta\ (\text{last}\ \Delta)]$$

- $\eta$ small $\Rightarrow$ learning like steepest descent
- Stabilizes oscillation, keeps downhill descent steady
- There are other schemes too; some update the momentum with training (these take extra tuning parameters).
- A popular approach is **ADADELTA** (Zeiler, 2012) . It updates “momentum” and “learning rate” like parameters as training progresses via only two hyper-parameters.

# Thoughts So Far

## So far:

1. Choose **initial weights** to be small and **standardize inputs**.

2. Looks like **ReLu** is a better choice for activation function than say logistic

3. With a few trials, look at the training error to choose a good **learning rate**.

**4.** **Regularizing** a more complex model is better than trying to guess a simple one.

**5.** **Momentum** is good too.

# It is Time to Talk About Computing the Gradient via… BackProp

# OS4118
# Week 6

Agenda:

- More on fully connected neural networks
  - Especially “**backprop**” fitting
- Special architectures: convolutional and recurrent neural networks
- If time permits, On to Text Data!

# Back Propagation

- A method for computing the gradient of the error function, evaluated at the mth values of the weights.

$$\left.\frac{\partial E}{\partial \mathbf{w}}\right|_{\mathbf{w}^m}$$

- It relies on the chain rule and the special structure of neural networks.
- Understanding Backprop can aid in intuition.
- After the *m*th step: “feed-forward” your training data through the NN using your weights $\boldsymbol{w}^m$. Keep track of the output from each of the hidden nodes and the outputs of the output nodes (y-hat’s). These values will be used in the gradient computation.

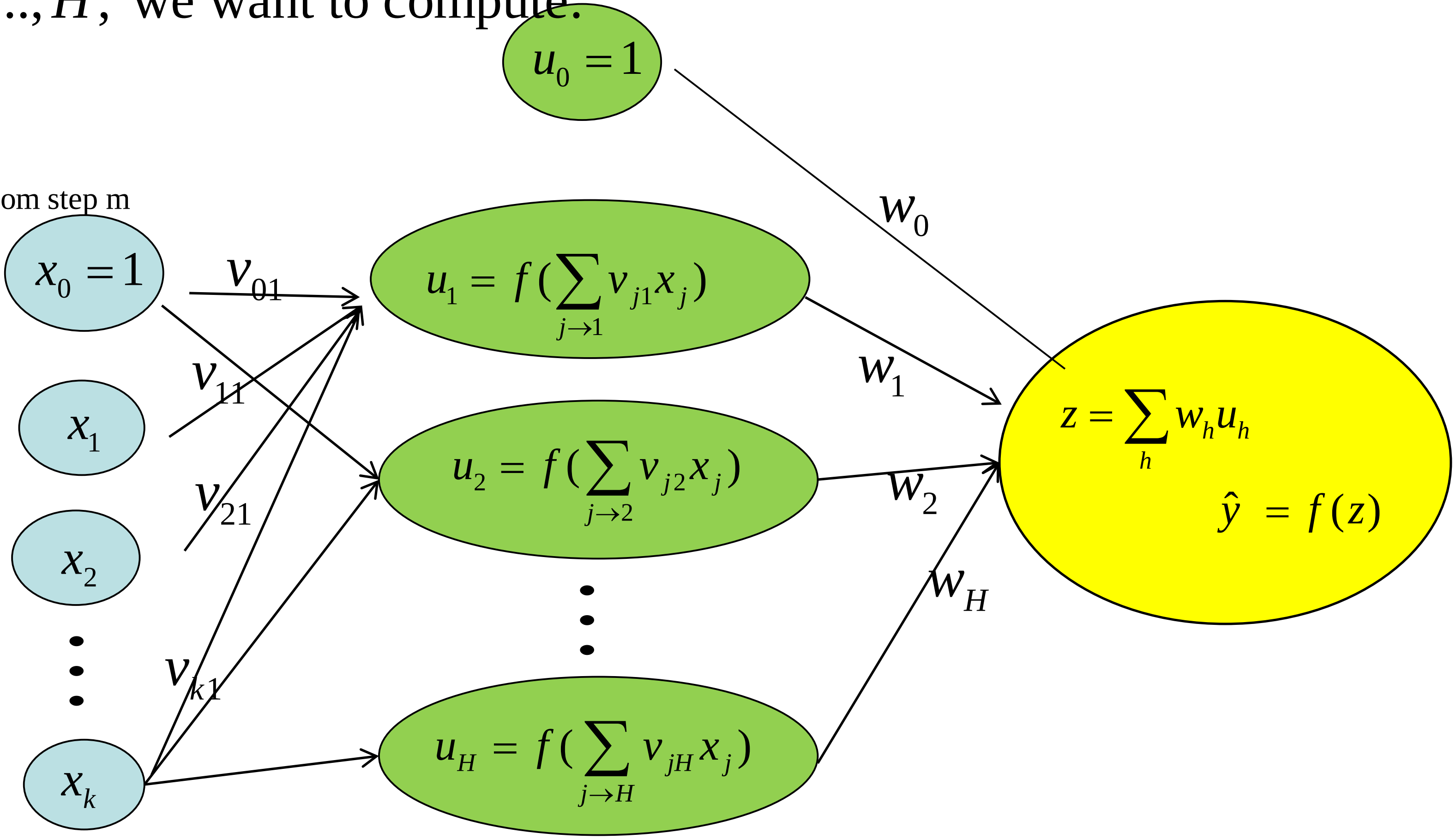
# Back Propagation

- It's all about the chain rule, e.g. for a NN with a single numeric target: $$\frac{\partial E}{\partial w_{43}} = \frac{\partial}{\partial w_{43}} \sum_{i=1}^{n} \left(y_i - \hat{y}_i\right)^2$$
$$= \sum_{i=1}^{n} \frac{\partial}{\partial w_{43}} \left(y_i - \hat{y}_i\right)^2$$
$$= \sum_{i=1}^{n} \frac{\partial}{\partial \hat{y}_i} \left(y_i - \hat{y}_i\right)^2 \frac{\partial \hat{y}_i}{\partial w_{43}}$$
$$= -2 \sum_{i=1}^{n} \left(y_i - \hat{y}_i\right) \frac{\partial \hat{y}_i}{\partial w_{43}}$$
$$= \ldots \;\; \textit{more chain rule...}$$
- Notice that, to evaluate this partial derivative at ***w***$^m$, (right after the *m*th step), we need the y-hats from the feeding the training set forward through the NN so far.
- To make my notation easier, think about evaluating the summand for one observation.

# Gradients for weights going to the output node.

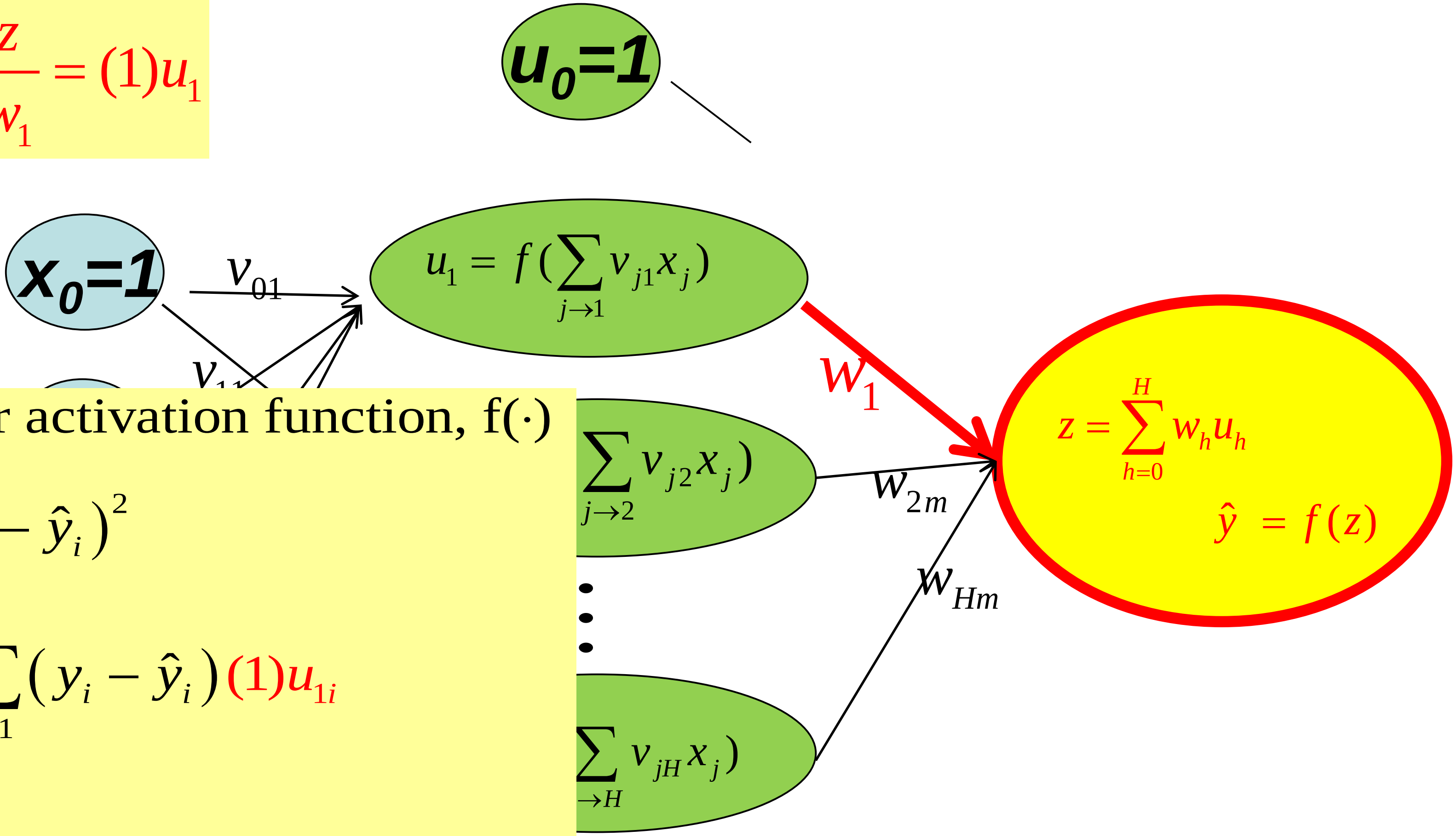
For $h = 0, \ldots, H,$ we want to compute:

$$\left.\frac{\partial \hat{y}}{\partial w_h}\right|_{\text{weights from step m}}$$



# E.g. update $w_1$ (with linear activation function)

$$\frac{\partial \hat{y}}{\partial w_1} = \frac{\partial \hat{y}}{\partial z}\frac{\partial z}{\partial w_1} = (1)u_1$$



$u_0=1$
$x_0=1$
$v_{01}$
$v_{11}$
$u_1 = f(\sum_{j\to 1} v_{j1}x_j)$
$\sum_{j\to 2} v_{j2}x_j)$
$\sum_{\to H} v_{jH}x_j)$
$w_1$
$w_{2m}$
$w_{Hm}$
$z = \sum_{h=0}^{H} w_h u_h$
$\hat{y} = f(z)$

With linear activation function, f(·)

$$E = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

$$\frac{\partial E}{\partial w_1} = -2\sum_{i=1}^{n}(y_i - \hat{y}_i)(1)u_{1i}$$

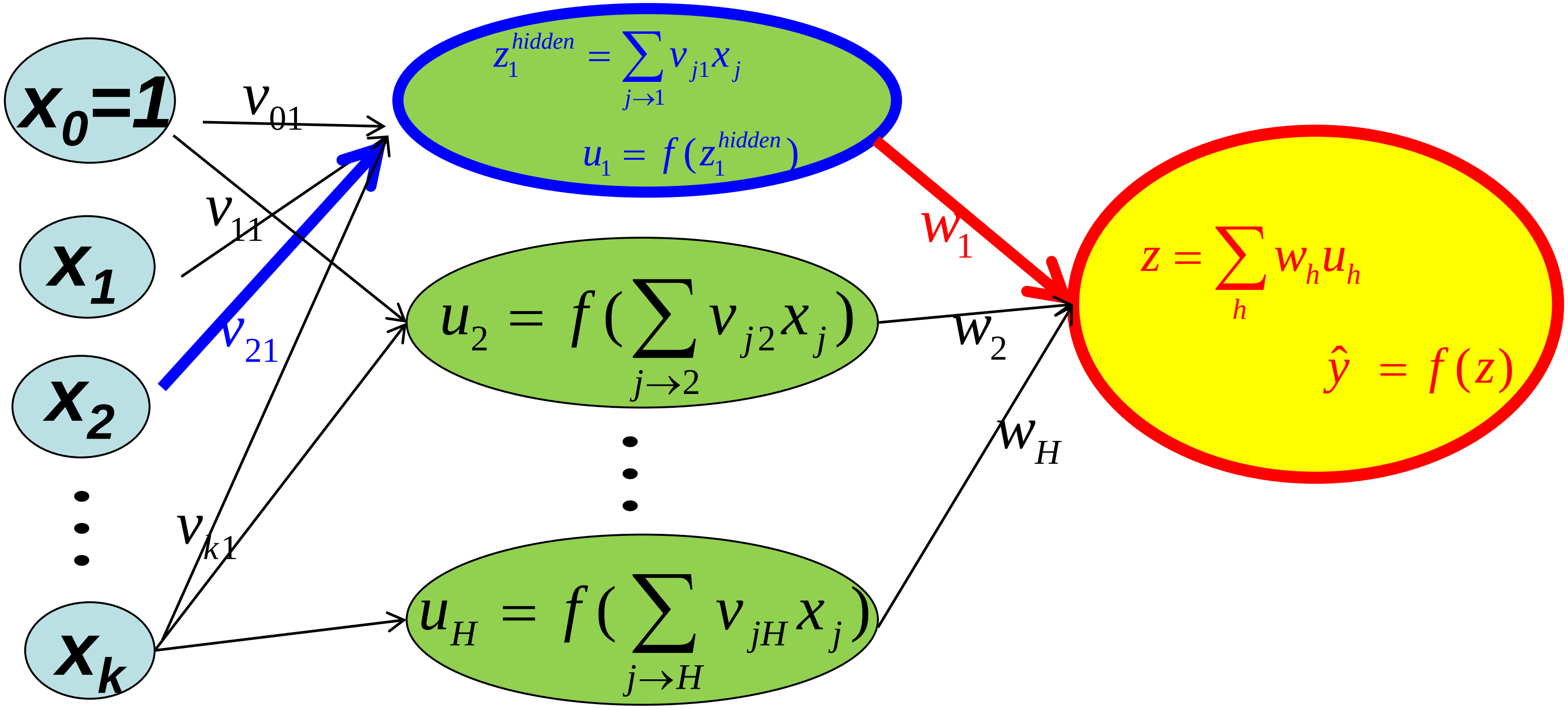
and

$$w_1^{m+1} \leftarrow w_1^{m} + \alpha \sum_{i=1}^{n}(y_i - \hat{y}_i)u_{i1}\Big|_{\mathbf{w}^m}$$

# Back Propagate to update the weights between the next set of layers. E.g. update $v_{21}$

$$\frac{\partial \hat{y}}{\partial v_{21}} = \frac{\partial \hat{y}}{\partial z}\frac{\partial z}{\partial u_1}\frac{\partial u_1}{\partial z_1^{hidden}}\frac{\partial z_1^{hidden}}{\partial v_{21}} = (1) w_1 f'(z_1^{hidden}) x_2$$

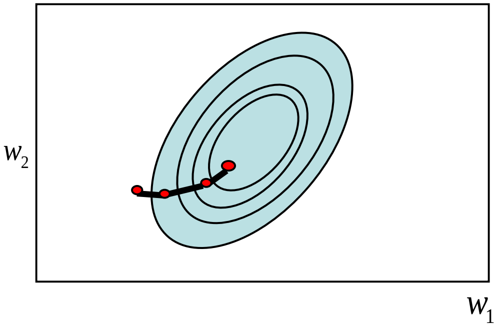
We already have these from the updates of the w's.



# When to update weights?

- "Batch" : present all examples (in the training set), compute the gradient of *E*, adjust weights, repeat. This can be slow; take memory.

$$\left.\frac{\partial E}{\partial \mathbf{w}}\right|_{\mathbf{w}^m} = \left.\frac{\partial}{\partial \mathbf{w}}\sum_{i=1}^{n}\left(y_i - \hat{y}_i\right)^2\right|_{\mathbf{w}^m}$$

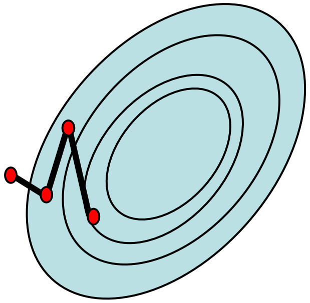


- But  with *n* large, do we really need all of the observations? Can't we approximate the gradient, using a **sample** of the observations? (Answer: Sure, why not?)
- Here is a crazy thought, what if we just use **one** observation to estimate the gradient?

# Stochastic Gradient Descent SGD in NN

- “On-line” or “Mini-batch” **approximate** the gradient of E using only one or a few observations.
- Weights are updated after each example or subset is presented. Very fast. Take noisy steps, and this can be an advantage and you can take more of them.
- SGD is sometimes taken to mean one observation, but in fact any mini-batch size yields a stochastic gradient.

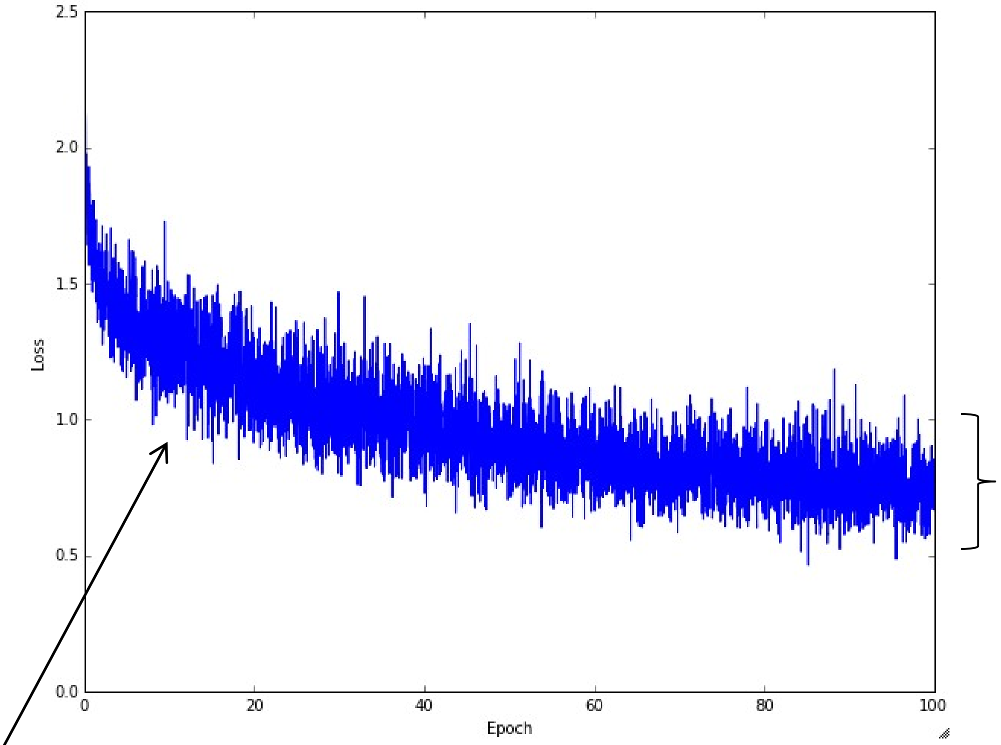
$$\frac{\partial E}{\partial \mathbf{w}} \approx \frac{\partial}{\partial \mathbf{w}} \sum_{i \in MiniBatch} \left( y_i - \hat{y}_i \right)^2$$



# Batch vs. On-line/Mini-batch

- Can help with memory storage problems. Can converge fast if the data have lots of similar obs.
- Observations presented in random order or as they appear in time.
- A training **epoch** is when all observations in a batch have been presented. Epochs can be repeated.
- At each epoch, examples may be randomized, training parameters e.g. learning rate, momentum, etc. may be changed.

# Baby-sitting Training: Actual Plot



A lot of variability in the loss from iteration to iteration indicates small batch size

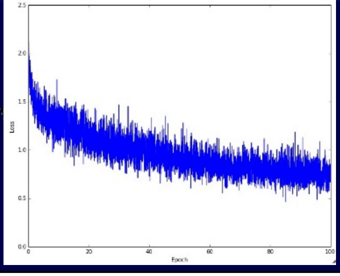
Or maybe we didn't randomize the observations?

Looks like exponential decrease, but maybe the learning rate is a bit high.

## Now for a fresh new idea to help with training…

## (OK, it’s 5 years old)

# But....

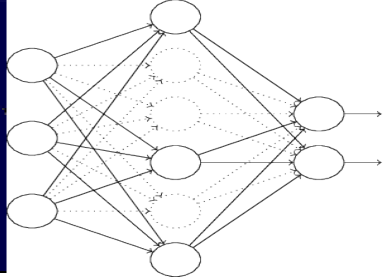


- On-line gradients, as estimates of the training set gradient, have **high variance**, we can decrease the variance by making larger mini-batch sizes (because averages of more things are better than averages of fewer things).
- OK, fine, use batch sizes of 100.

**And**

- Take advantage of the high-variance, like Hinton et. al (2012) – remember **ensembles**?
- Make those 100 contributions to the gradient even more **independent-ish** and average them. Then the estimate of the gradient should be even better.

# Dropout (Hinton et.al., 2012)

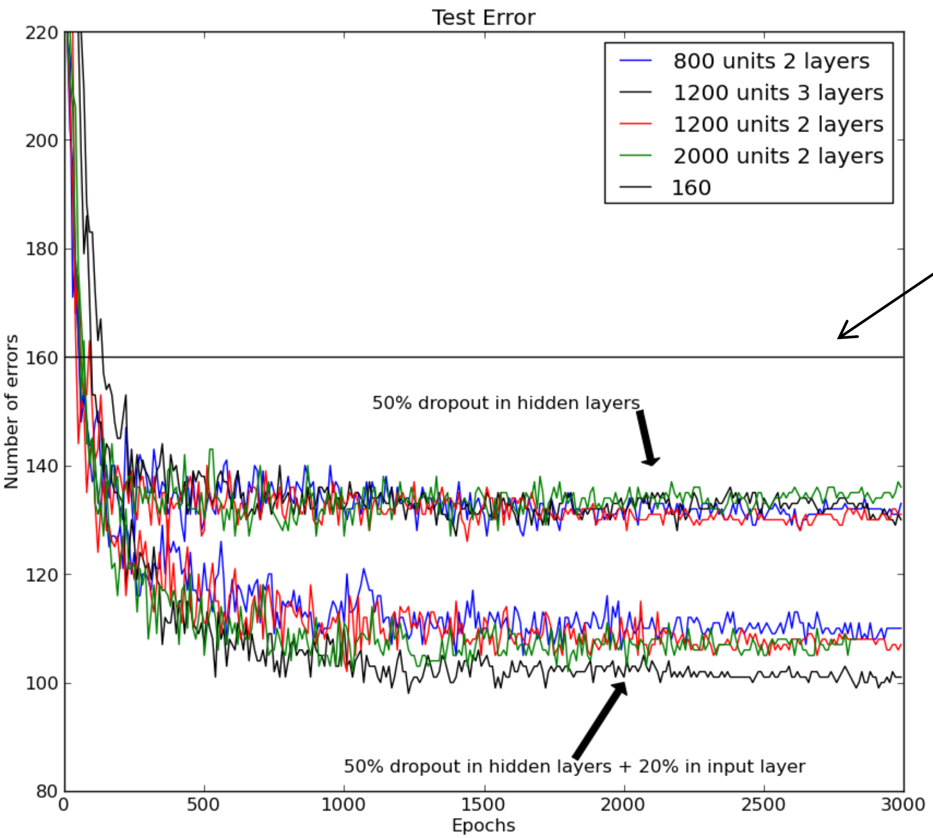


- After the $m^{th}$ step, select 100 observations (examples) .
- Hinton borrows an idea from Random Forests. For each example, **drop out** each hidden node with $p_1 = 50\%$ chance, and each input node with $p_2 = 20\%$ chance (i.e. set weights coming from those nodes to zero).
- Feed each example through its (crippled) NN; compute gradient of $E$ (based on the single case) for weights whose nodes are present.
- Average each weight's gradients over examples where that node is present. Then update weights.
- Optional: regularize as follows, if at any update,

$$\Sigma\, w^2 > c \quad \text{then} \quad w \leftarrow w/c^{1/2}$$

- After training, when predicting or testing, use the entire network and multiply the trained weights by $p_1$

# Dropout of the input units help when input features are noisy.

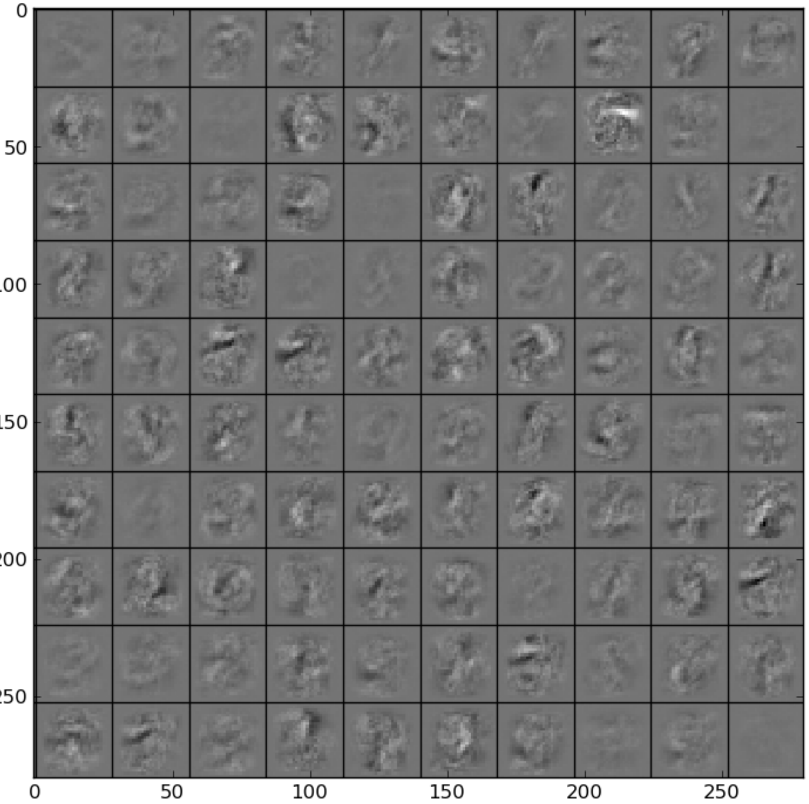


In 2012, fully connected with drop out did better than the best (CNN) to date. 160/10000 = .016

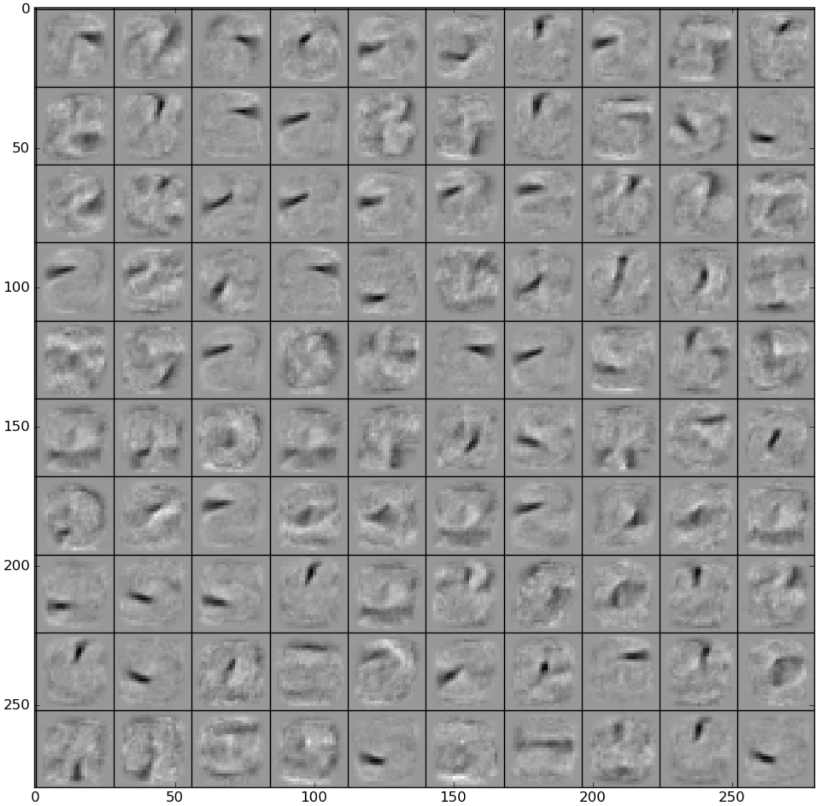
MNIST n = 60,000; **Mini-batchs = 100**
3000 epochs
Learning rate = 10 decreased by a factor of .998 at each epoch.
**No weight decay, but weights rescaled to have squared length < 15.**
Momentum = .5 then increased linearly to .99 for 500 epochs then stays at .99 for the remaining epochs.
**Initial weights Normal( 0, 0.01 )**

# Hinton et. al. (2012) says with dropout,  features don't "co-adapt"

No Dropout



Dropout

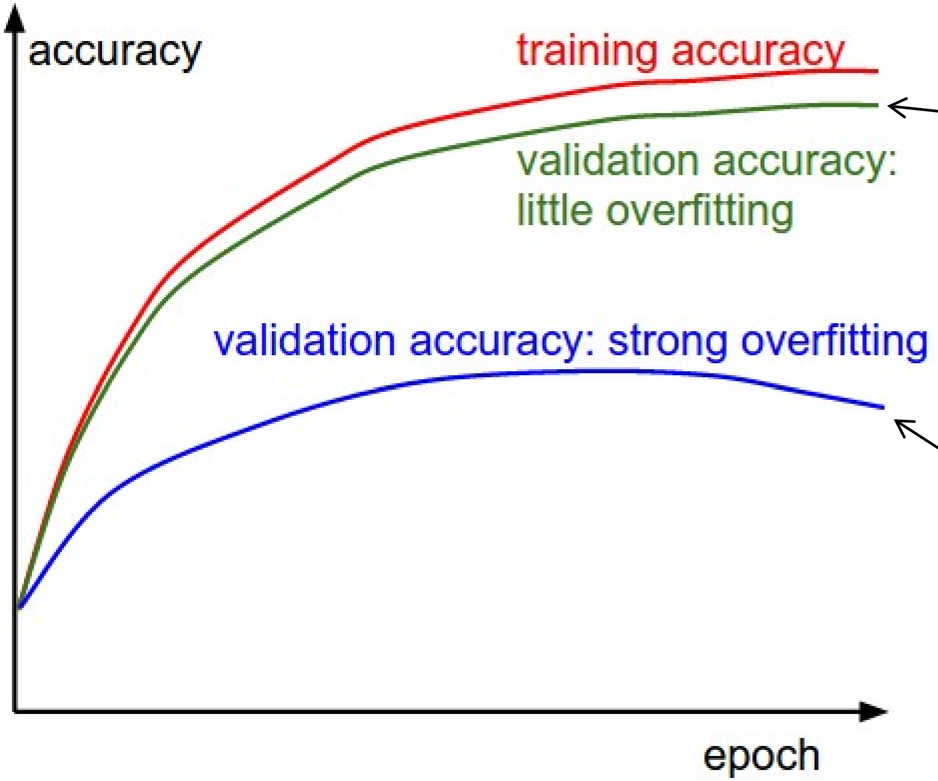


100 "learned" features taken from the first layer of a 784 – 500 – 500  – 10  network. Each picture shows 28x28 image that maximally activates a node in the first hidden layer (MNIST data)

# Dropout Example

- Simple addition to Keras flow…
- …Although in our example it doesn’t really help
- Best suited for the sorts of huge models we might fit in real life
- All in the service of preventing over-fitting

# Baby-sitting Training:
# Training vs. Validation (or Cross-Validation)



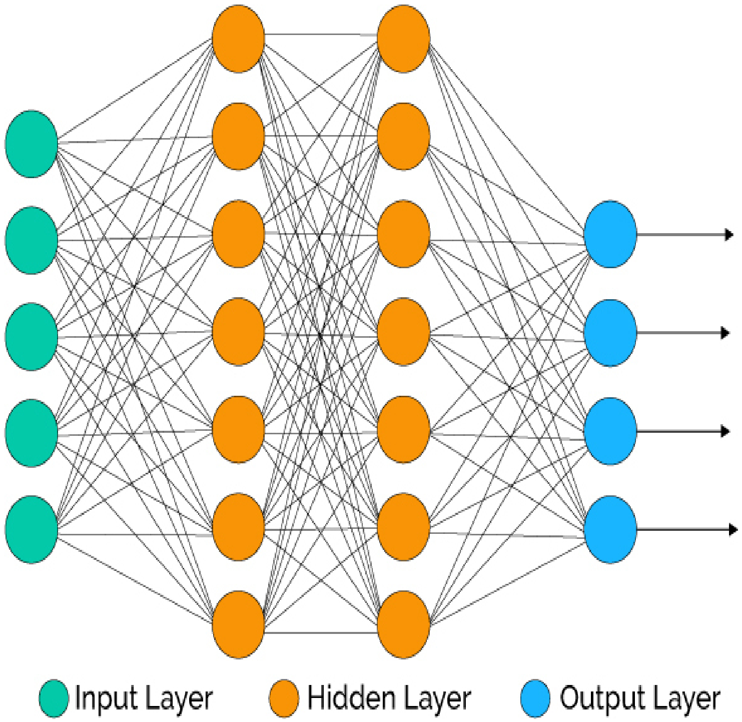
If these are tracking too closely, you may be under-fitting. Increase the complexity of the model.

If validation accuracy goes down, increase regularization

We need to keep an eye on the gap between training and validation accuracy.

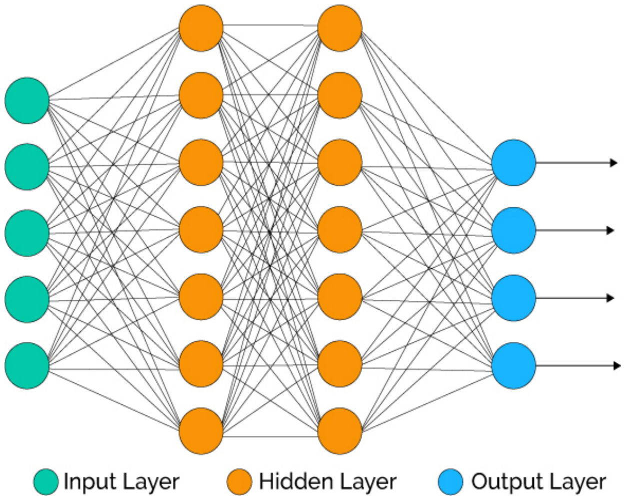
# Neural Networks can "Learn"

- Train voice, hand-writing recognition, etc. networks on a training set from the **general population**. Then update weights based on person-specific examples with a gradient descent-like algorithm (this is what your iPhone does).



# Transfer Learning

- And, then there is **transfer** learning where a network is trained with lots of data (not necessarily like your data). Here is one possibility:
  - Kolsch wants to classify images with and without AK-47s. Start with a general ImageNet trained Deep NN which classifies 1000 images into 1000 types (bird, snowplow, etc.).
  - Throw Kolsch's images through the ImageNet NN, use its 1000 output nodes as inputs for the new NN.



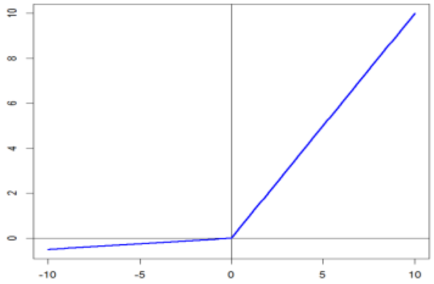
# Other approaches for increasing the effective data set size

- For vision tasks, increase *n* by taking mirror images, rotating images, ….(augmenting the data)

- There is a whole literature on pre-training weights using **unsupervised data** e.g. untagged images (Andrew Ng).

- This ‘semi-supervised’ training is a big deal when there are very few tagged examples or perhaps the tagged examples are not quite of the right sort e.g. cyber security.

# To Avoid Flat Spots:



- **Standardize numeric  inputs**.
  - Often inputs are **centered at 0**.
  - After training, subsequent data (validation sets, test sets, predictions for new cases) are standardized **with the values used to standardize the training set**
- **Choose small initial weights.**
  - **The old way**:  Uniform[-0.7,0.7] , or Normal(0,0.01)  (Hinton , 2012).
  - **A newer way:** Uniform[-r, r]  where r depends on the number of weights (He et. al., ,2015)  (Glorot and Bengio, 2010)
  - **Or:** Sparse initialization ( Martins, 2010). Set most weights to be zero and randomly select some to be small.
  - Small starting weights make the logistic activation function nearly linear and hence give a **less complex model**.  As |weights| become large, introduce nonlinearity and more complexity.
- **Regularize weights** ( $L_2$-Norm or ridge penalty is called weight decay)
- **Replace logistic and hyperbolic tangent activation functions** with ReLU  or Leaky ReLU or other activation function (He, et. al. 2015)

# Software:

http://cs231n.stanford.edu/slides/winter1516_lecture12.pdf

- Caffe
- Torch
- Theano
- CNTK (Microsoft Cognitive Toolkit [open source])
- TensorFlow (a Python Package)
  - From Google; very similar to Theano
  - all about computation graphs
  - Easy visualizations (TensorBoard)
  - Multi-GPU and multi-node training
- h2o Deepwater; uses TensorFlow, Caffe,… (Not available on our AWS instance)

# Software:

http://cs231n.stanford.edu/slides/winter1516_lecture12.pdf

- Caffe
- Torch
- Theano
- CNTK (Microsoft Cognitive Toolkit [open source])
- TensorFlow (a Python Package)
  - From Google; very similar to Theano
  - all about computation graphs
  - Easy visualizations (TensorBoard)
  - Multi-GPU and multi-node training
- h2o Deepwater, uses TensorFlow, Caffe,… (Not available on our AWS instance)

Keras supports…