COMS E6998 010

Practical Deep Learning Systems Performance

Lecture 3 09/24/20

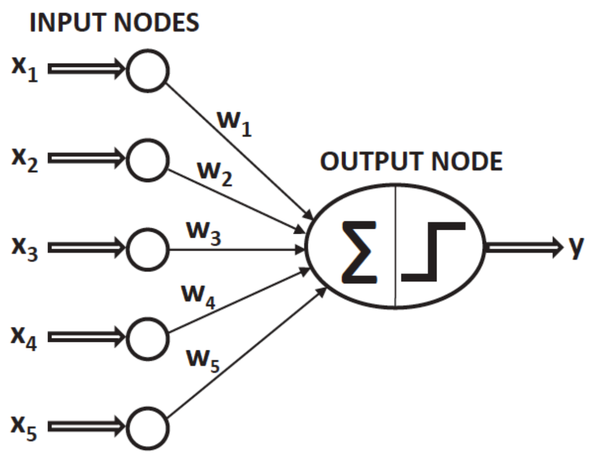
Logistics

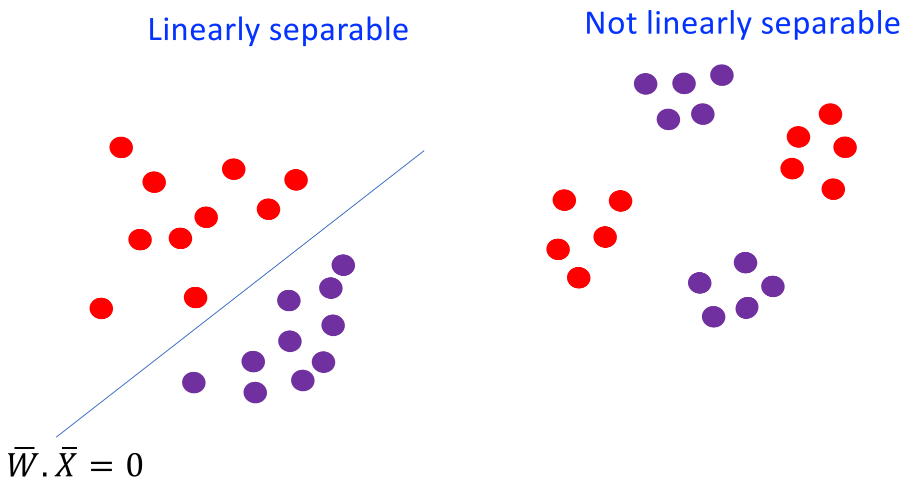
- Reading-1 due Sept. 28 by 11:59 PM
- Homework-1 due Oct. 1 by 11:59 PM
- Late submissions not allowed
- Office Hours:
  - Parijat Dube: Fridays 4 PM - 6 PM
  - Brandon Liang: Mondays 10 AM -12 PM
  - Jianqiao Hao: Thursdays 4M - 6PM
- Seminar: 6-9 PM on Nov. 2, 4, and 6. Sign-up sheet will be posted.
- Project proposals due Oct. 29

Recall from Last lecture

- Bias-variance tradeoff
- Linear separability
- Generalization and Cross-validation
- Regularization techniques in ML and DL
- Performance metrics
- Universal Approximators Theorem; Depth vs Width
- Dataset augmentation, Weight decay, Early stopping, Dropout

Single Layer Perceptron



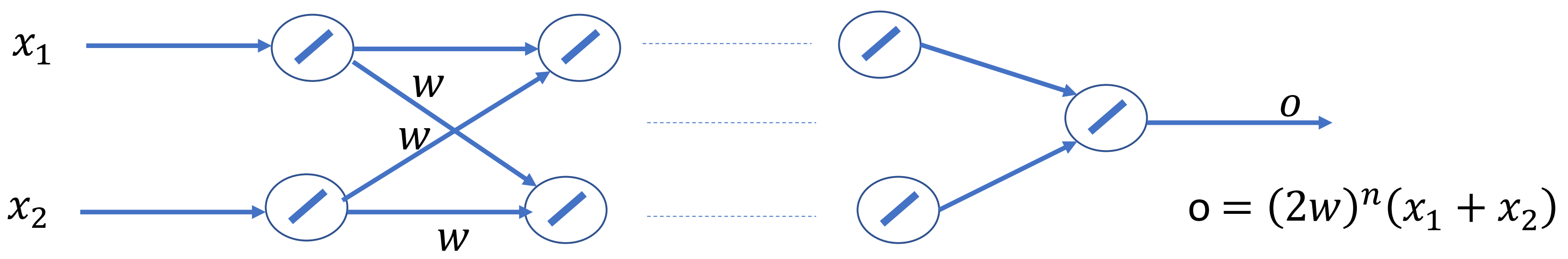


$$\hat{y} = \text{sign}\{\overline{W} \cdot \overline{X}\} = \text{sign}\{\sum_{j=1}^{d} w_j x_j\}$$

$$\overline{W} \Leftarrow \overline{W} + \alpha \underbrace{(y - \hat{y})}_{\text{Error}} \overline{X}$$

Perceptron training uses one training data at each update
One cycle through the entire training data set is referred to as an epoch ⇒ Multiple epochs required
Perceptron weight updates are not gradient descent as loss function is not differentiable
Perceptron training will not converge for not linearly separable dataset

How about adding more layers ?



$o = (2w)^n(x_1 + x_2)$

Multi-layer neural network with linear activation functions ⇔ single layer neural network with linear activation

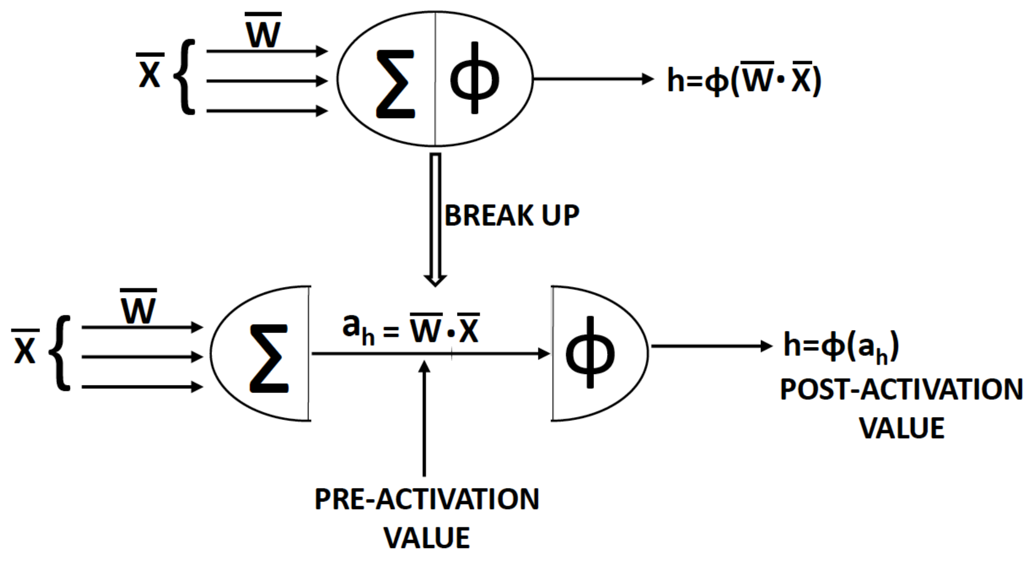
- A neural network with any number of layers but only linear activations can be shown to be equivalent to a single-layer network
  - True for any activation function in the output node

- Cannot solve XOR problem

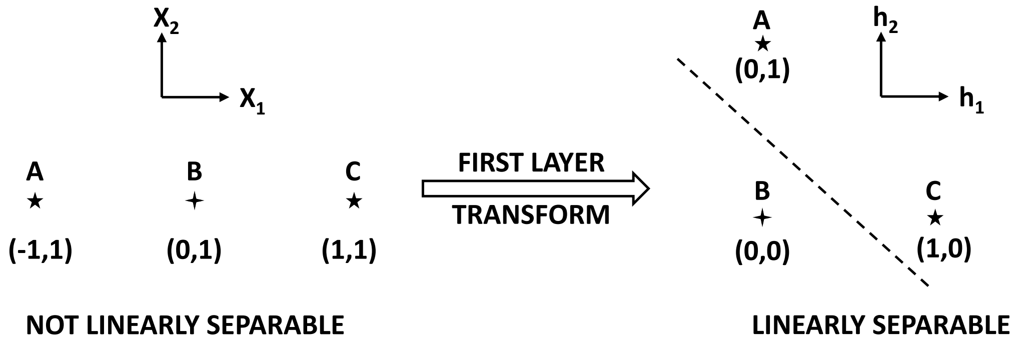
x_1	x_2	u
0	0	0
0	1	1
1	0	1
1	1	0

$$0w_1 + 0w_2 + b \le 0 \iff b \le 0$$
$$0w_1 + 1w_2 + b > 0 \iff b > -w_2$$
$$1w_1 + 0w_2 + b > 0 \iff b > -w_1$$
$$1w_1 + 1w_2 + b \le 0 \iff b \le -w_1 - w_2$$

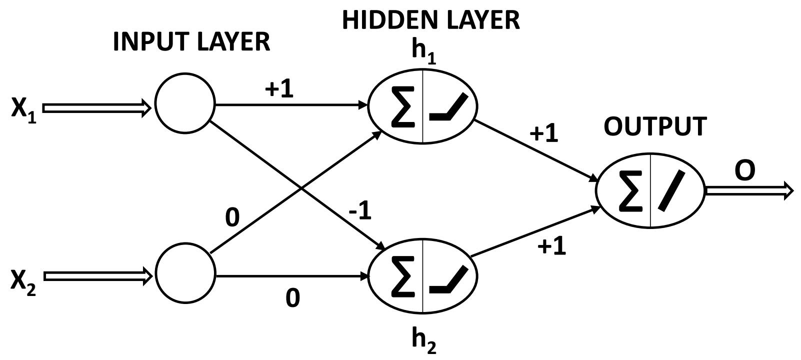
Bringing Non-linearity with Activation Functions



Non-linear activations in hidden layers

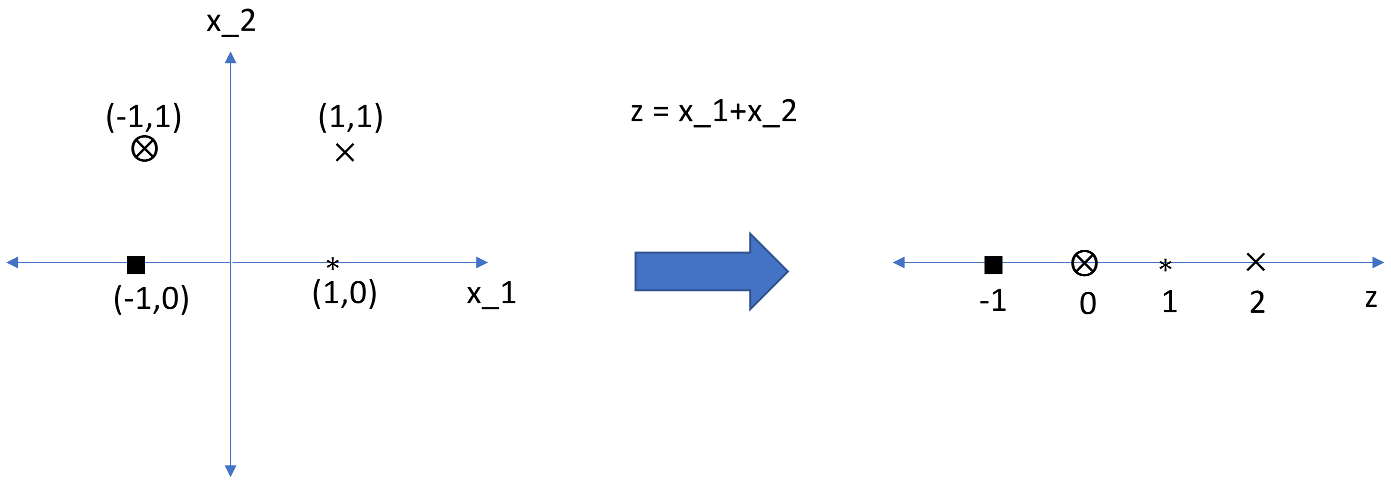


$$h_1 = \max\{x_1, 0\} \quad h_2 = \max\{-x_1, 0\}$$



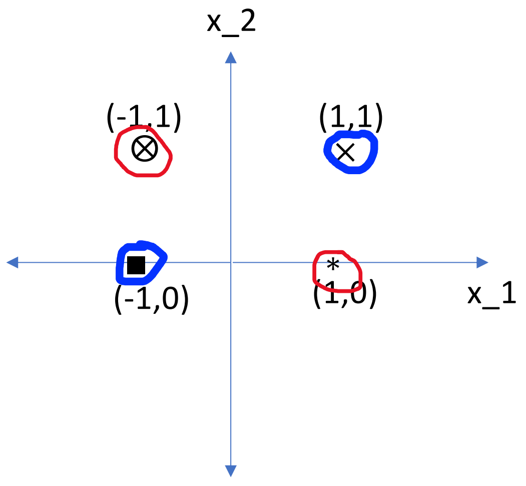
X1	X2	h1	h2	h1+h2
-1	1	0	1	1
0	1	0	0	0
1	1	1	0	1

2D to 1D transformation example

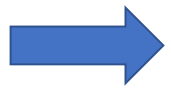


2D to 1D transformation example

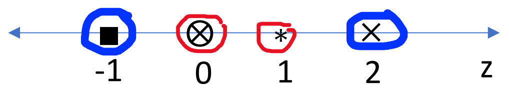
Points in 2-D are not linearly separable



z = x_1+x_2

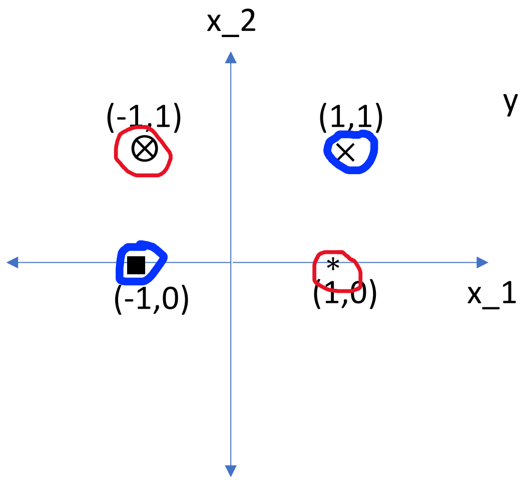


Points in 1-D are not linearly separable

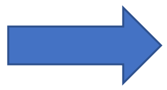


2D to 1D transformation example

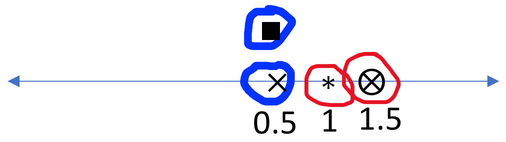
Points in 2-D are not linearly separable



y = max(x_1,x_2)-0.5*min(x_1,x_2)

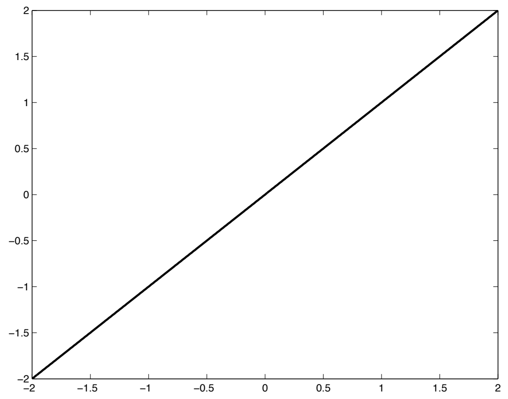


Points in 1-D are linearly separable

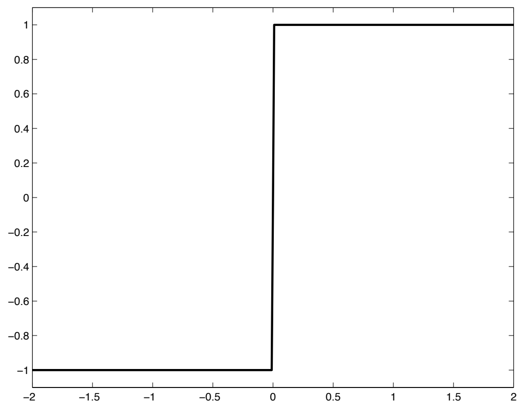


Activation functions

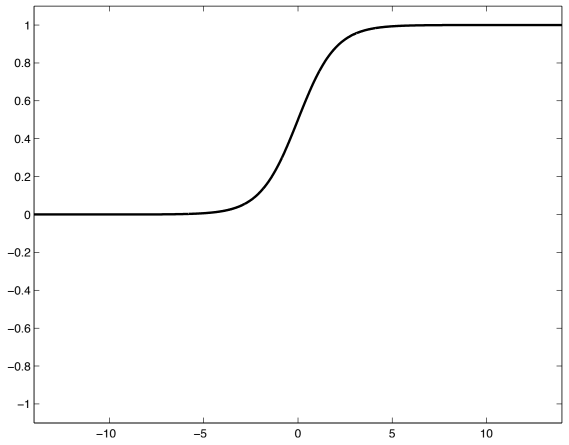
Linear $\phi(z) = z$



Sign $\phi(z) = sign(z)$

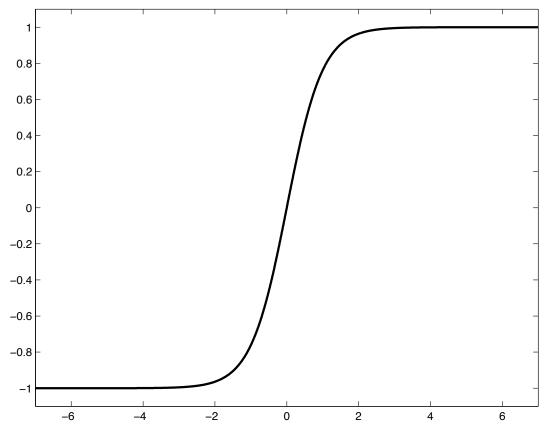


Sigmoid $\phi(z) = \frac{1}{1+e^{-z}}$

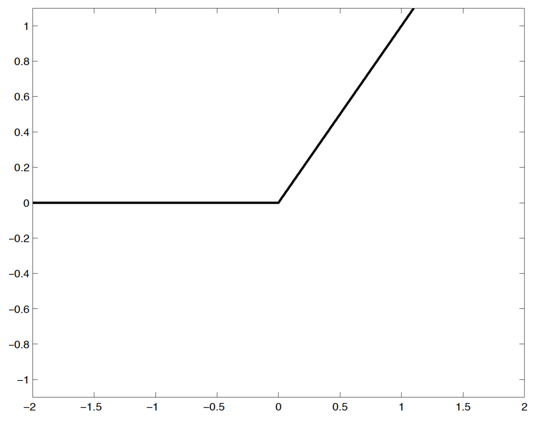


Activation functions

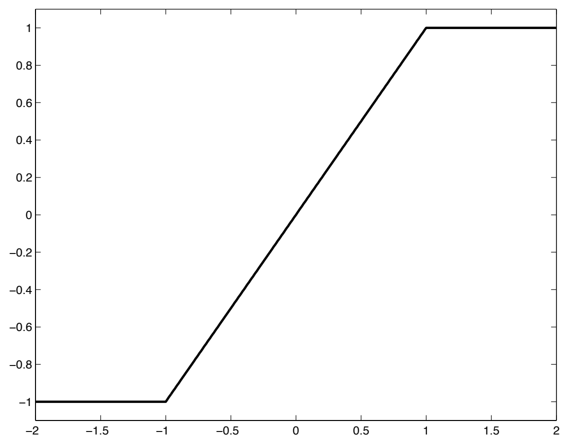
Tanh $\phi(z) = \frac{e^{2v}-1}{e^{2v}+1}$



ReLU (Rectified Linear Unit)
$\phi(z) = \max\{z, 0\}$



Hard Tanh
$\phi(z) = \max\{\min[z, 1], -1\}$



- Also called *squashing* functions
- tanh(z) = 2.sigmoid(2z)-1
- ReLU is most common for hidden layers;

Activation Functions

- An activation function Φ(v) in the output layer can control the nature of the output (e.g., probability value in [0, 1])
- In multilayer neural networks, activation functions bring nonlinearity into hidden layers, which increases the complexity and representation power of the model
- Continuous, differentiable activation functions for gradient descent updates (need derivative of activation functions in weight updates during training)

Loss Functions

- Form of loss functions depends on the type of output (continuous valued or discrete) and on the range of output values
- Least-squares regression for continuous valued targets
  - Least-square regression loss
    - Linear activation in output

$$Loss = \ (y - \hat{y})^2$$

- Probabilistic class prediction for discrete valued targets
  - Logistic regression loss
    - Sigmoid activation in output
    - If y is binary valued in {-1,1} and $\hat{y} \ \in (0,1)$

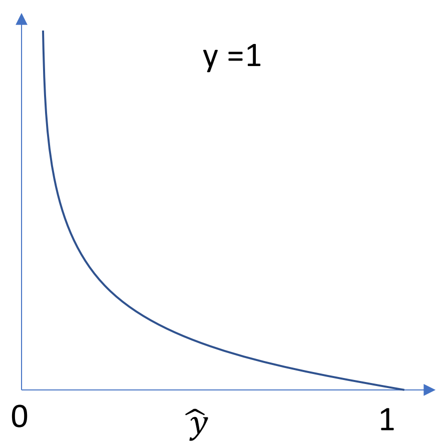
$$Loss = -\log\left|\frac{y}{2} - 0.5 + \hat{y}\right|$$

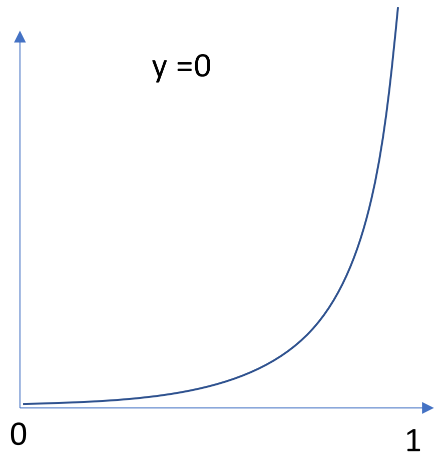
    - If y is binary valued in {0,1} and $\hat{y} \ \in (0,1)$

$$Loss = -\mathrm{y}\log\hat{y} - (1-y)\log(1-\hat{y})$$

Logistic Regression Loss Function

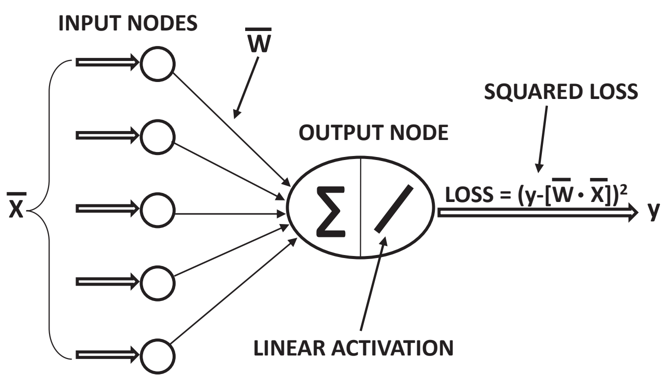
$$Loss = -\mathrm{y} \log \hat{y} - (1-y) \log(1-\hat{y})$$



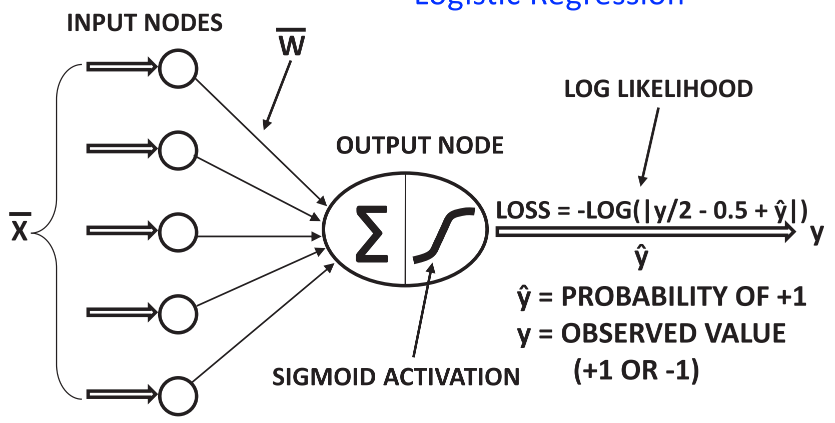


Neural Networks for Machine Learning Models

Linear Regression



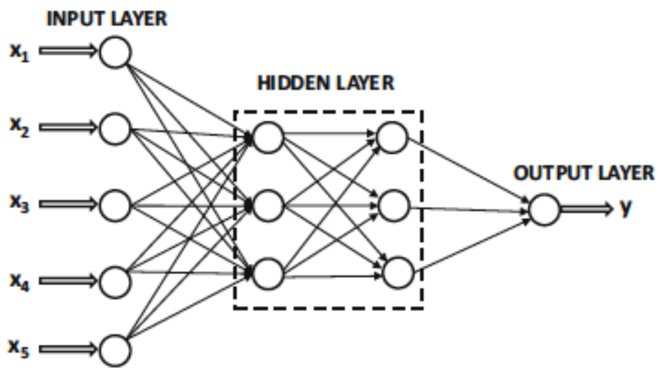
Logistic Regression

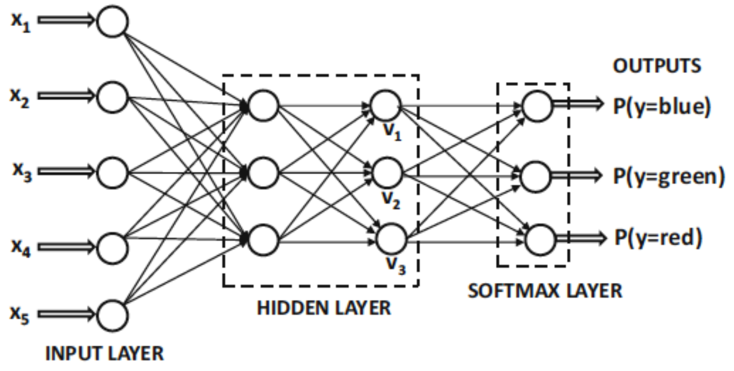


$$\overline{W} \Leftarrow \overline{W} + \alpha(y - \hat{y})\overline{X}$$

$$\overline{W} \Leftarrow \overline{W} + \alpha \frac{y_i \overline{X_i}}{1 + \exp[y_i(\overline{W} \cdot \overline{X_i})]}$$

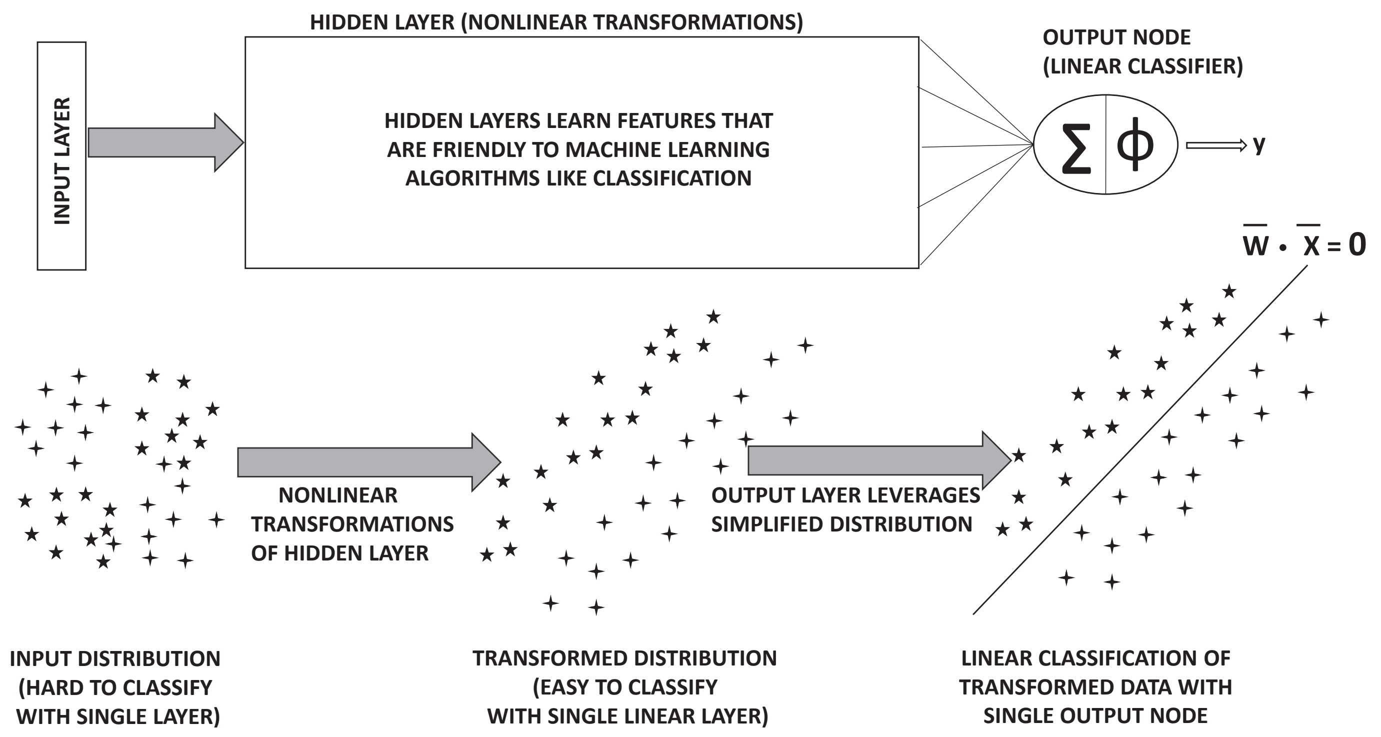
Multilayer Neural Networks





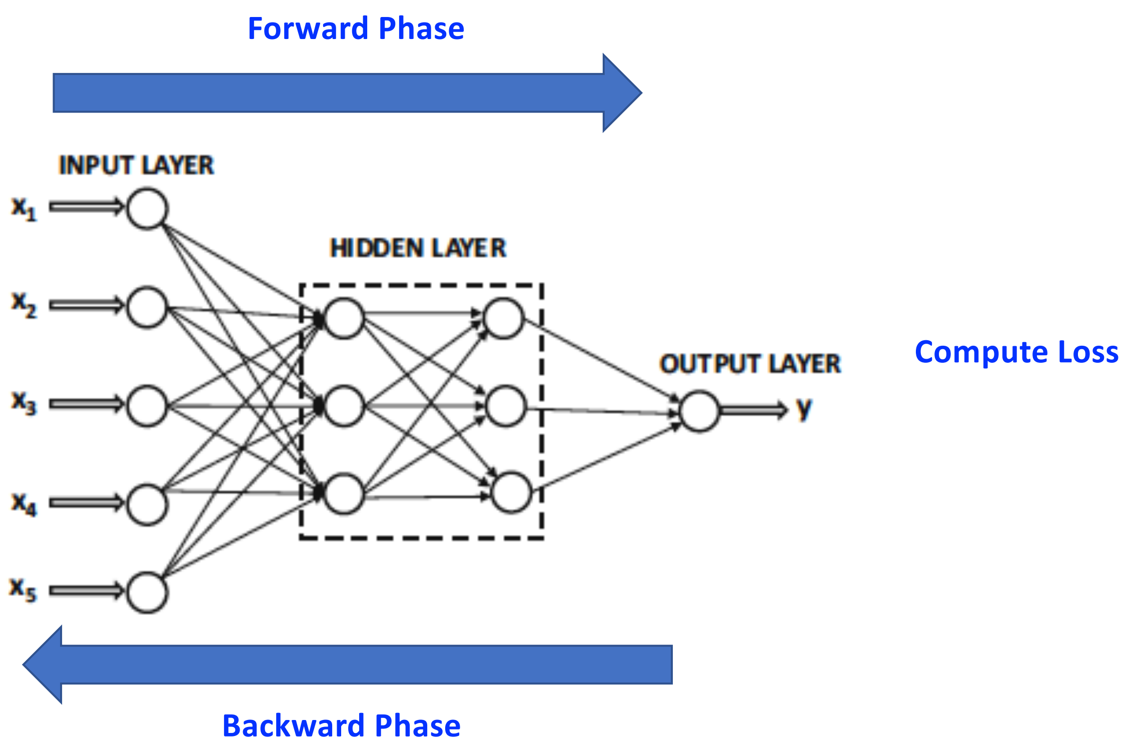
Hidden Layers Role

Hierarchical Feature Engineering


HIDDEN LAYER (NONLINEAR TRANSFORMATIONS)
INPUT LAYER
HIDDEN LAYERS LEARN FEATURES THAT ARE FRIENDLY TO MACHINE LEARNING ALGORITHMS LIKE CLASSIFICATION
OUTPUT NODE (LINEAR CLASSIFIER)
Σ
Φ
y
$\overline{W} \cdot \overline{X} = 0$
NONLINEAR TRANSFORMATIONS OF HIDDEN LAYER
OUTPUT LAYER LEVERAGES SIMPLIFIED DISTRIBUTION
INPUT DISTRIBUTION (HARD TO CLASSIFY WITH SINGLE LAYER)
TRANSFORMED DISTRIBUTION (EASY TO CLASSIFY WITH SINGLE LINEAR LAYER)
LINEAR CLASSIFICATION OF TRANSFORMED DATA WITH SINGLE OUTPUT NODE

Deep Learning Training

- Forward phase
- Loss calculation
- Backward phase
- Weight update



Deep Learning Training Steps

- Forward phase:
  - compute the activations of the hidden units based on the current value of weights
  - calculate output
  - calculate loss function
- Backward phase:
  - compute partial derivative of loss function w.r.t. all the weights;
  - use *backpropagation algorithm* to calculate the partial derivatives recursively
  - backpropagation changes the weights (and biases) in a network to decrease the loss
- Update the weights using gradient descent

Softmax Activation Function

- Function is calculated with respect to multiple inputs
- Converts real valued predictions into output probabilities

$$o_i = \frac{\exp(v_i)}{\sum_{j=1}^{k} \exp(v_j)} \quad \forall i \in \{1, \ldots, k\}$$

- Backpropagation with softmax
  - Always used in output layer, not in hidden layers
  - Always paired with cross-entropy loss

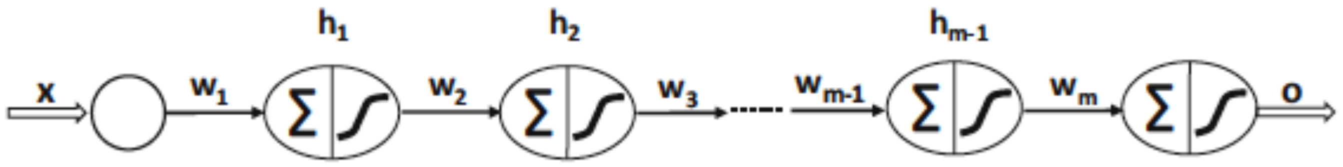
$$L = -\sum_{i=1}^{k} y_i \log(o_i) \qquad \frac{\partial L}{\partial v_i} = \sum_{j=1}^{k} \frac{\partial L}{\partial o_j} \cdot \frac{\partial o_j}{\partial v_i} = o_i - y_i$$

  - Derivatives needed for backpropagation have simple form

Hyperparameters in Deep Learning

- Network architecture: number of hidden layers, number of hidden units per later
- Activation functions
- Weight initializer
- Optimizer
- Learning rate
- Batch size
- Momentum

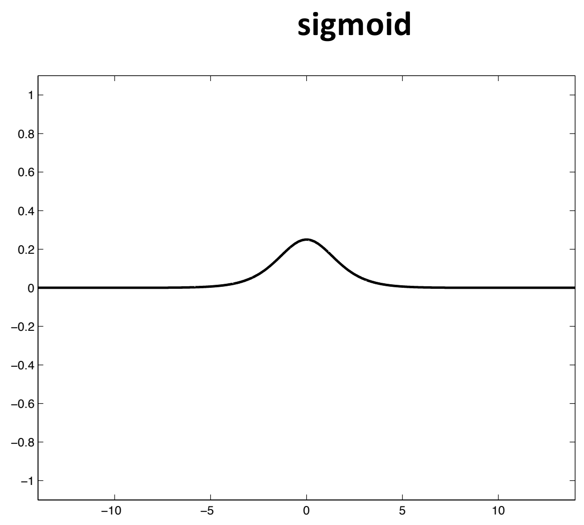
Vanishing and Exploding Gradients

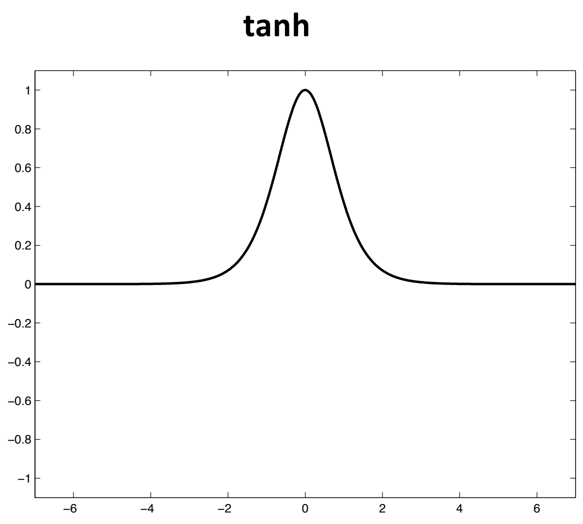


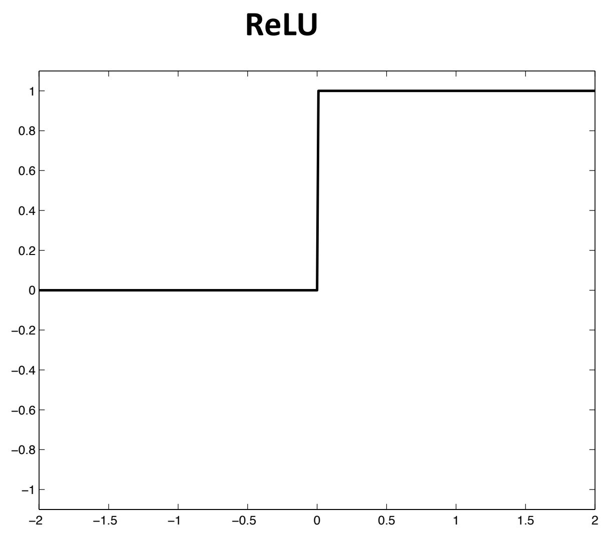
$$\frac{\partial L}{\partial h_t} = \phi'(w_{t+1}h_t).\,w_{t+1}.\frac{\partial L}{\partial h_{t+1}}$$

- For sigmoid activation, $\phi'(z) = \phi(z)(1 - \phi(z))$, has maximum value of 0.25 at ϕ(z)=0.5
- Each $\frac{\partial L}{\partial h_t}$ will will be less than 0.25 of $\frac{\partial L}{\partial h_{t+1}}$
- As we (back) propagate further gradient keep decreasing further; After r layers the value of gradient reduces to 0.25^r (= 10^{-6} for r=10) of the original value causing the update magnitudes of earlier layers to be very small compared to later layers => vanishing gradient problem.
- If we use activation with larger gradient and larger weights=> gradient may become very large during backpropagation (exploding gradients)
- Improper initialization of weights also causes vanishing (too small weights) or exploding (too large weights) gradients

Activation Functions Derivatives







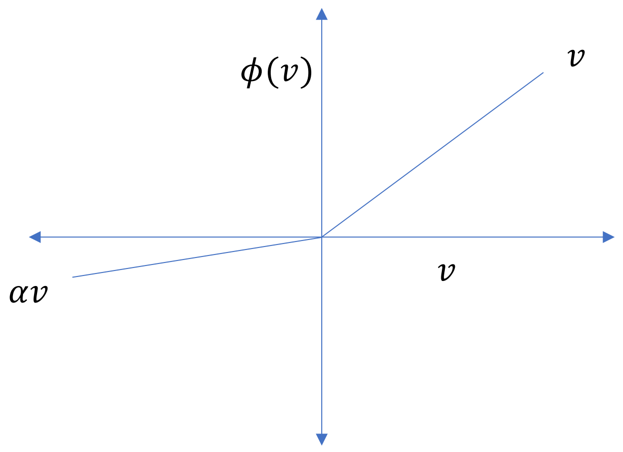
- Sigmoid and tanh derivatives vs ReLU
- Sigmoid and tanh gradients saturate at large values of argument; very susceptible to vanishing gradient problem
- ReLU is faster to train; most commonly used activation function in deep learning

Preventing vanishing gradients

- Use piece-wise linear functions like ReLU as activation. Gradients are not close to 0 for higher values of input.
- Piece-wise linear can cause dead neuron
    - Causes: improper weight initialization, high learning rates
    - Hidden unit will not fire for any input
    - Weights of the neuron will not be updated
- Leaky ReLU activation

$$\Phi(v) = \begin{cases} \alpha \cdot v & v \leq 0 \\ v & \text{otherwise} \end{cases}$$

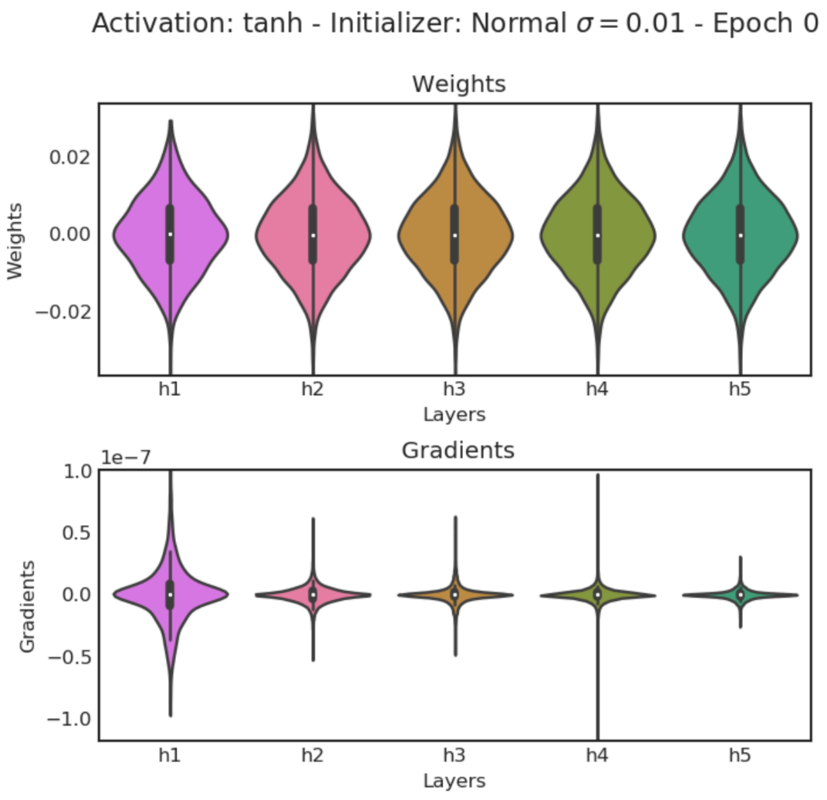
$$\alpha \in (0, 1)$$

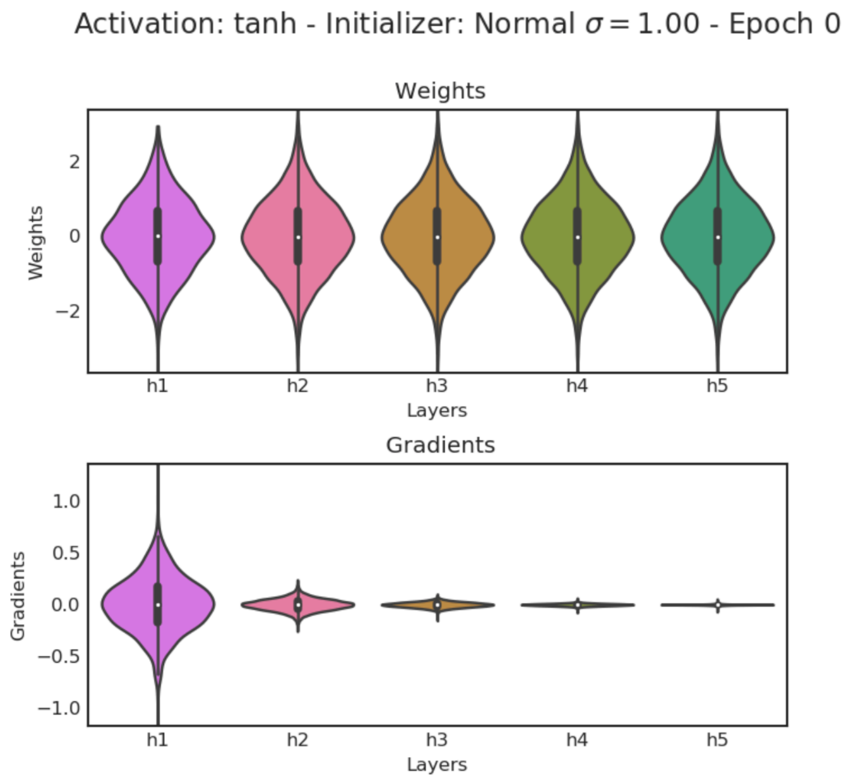


Weight Initialization

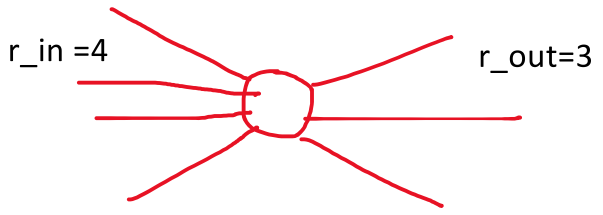
- Initializing all weights to same value will cause neurons to evolve symmetrically
- Generally biases are initialized with 0 values and weights with random numbers; Initializing weights to random values breaks symmetry and enables different neurons to learn different features
- Initial value of weights is important.
  - Poor initializations can lead to bad convergence or no learning.
  - Instability across different layers (vanishing and exploding gradients).

Vanishing and Exploding Gradients





Popular Weight Initializers



- Xavier/Glorot (Sigmoid or Tanh)

  Each neuron weight is sampled from 0 mean Gaussian distribution with standard deviation

$$\sqrt{2/(r_{in} + r_{out})}.$$

when r_{in} and r_{out} are number of input and output weights for the neuron

  - **Xavier initialization,** is also referred to as (like in *Keras*) **Glorot initialization**.

- He
- Sample weights from 0 mean Gaussian distribution with standard deviation

$(\sqrt{2/r}$ for ReLU)

r can be r_{in} or r_{out}

Normalizing Input Data

- Min-max normalization (for feature j of input datapoint i)

$$x_{ij} \Leftarrow \frac{x_{ij} - min_j}{max_j - min_j}$$

  - Data with smaller standard deviation; scaled to be in the range [0,1]
  - Lessen the effect of outliers
- Standardization

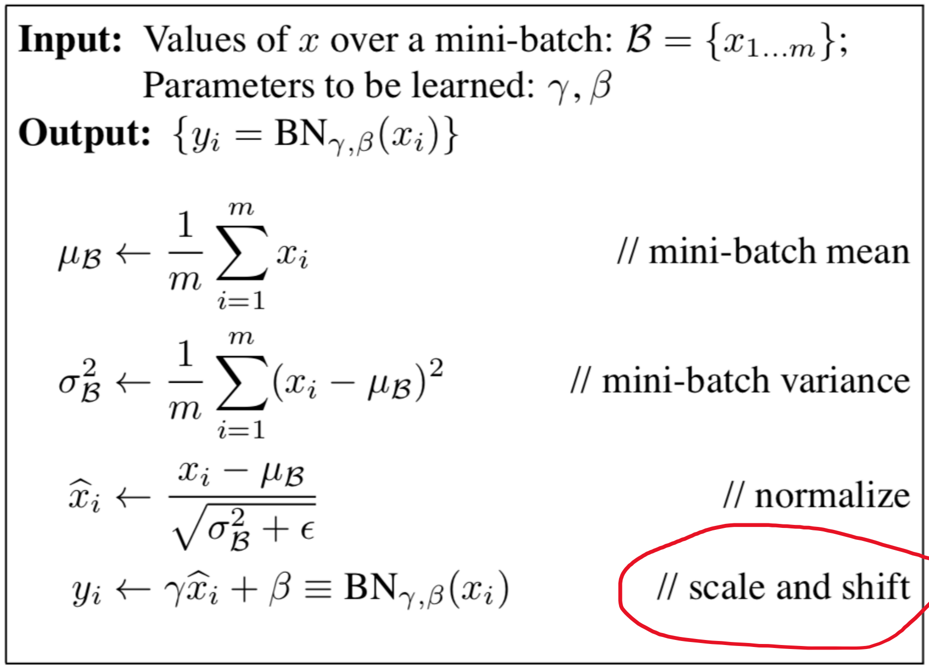
$$x_{ij} \Leftarrow \frac{x_{ij} - mean_j}{std\_dev_j}$$

- Normalization helps in the convergence of optimization algorithm
- Should apply same normalization parameters to both train and test set
- Normalization parameters are calculated using train data
- Training converges faster when the inputs are normalized

Batch normalization

- Internal covariance shift – change in the distribution of network activations due to change in network parameters during training
- Idea is to reduce internal covariance shift by applying normalization to inputs of each layer
- Achieve fix distribution of inputs at each layer
- Normalization *for each training mini-batch*.
- Batch normalization enables training with larger learning rates
  - Reduces the dependence of gradients on the scale of the parameters
  - Faster convergence and better generalization

Batch normalization



```
Input: Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
       Parameters to be learned: $\gamma, \beta$
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

  $\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$                                // mini-batch mean
  $\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2$                // mini-batch variance
  $\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$                              // normalize
  $y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$                 // scale and shift
```

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Why this step ?

Gradient Descent

$$L = \sum_{i=1}^{n} L_i$$

$$\overline{W} \Leftarrow \overline{W} - \alpha \frac{\partial L}{\partial \overline{W}}$$

- Loss is calculated over all the training points at each weight update
- Memory requirements may be prohibitive

Stochastic Gradient Descent (SGD)

$$\overline{W} \Leftarrow \overline{W} - \alpha \frac{\partial L_i}{\partial \overline{W}}$$

- Loss is calculated using one training data at each weight update
- Stochastic gradient descent is only a randomized approximation of the true loss function.

Mini-batch Gradient Descent

$$\overline{W} \Leftarrow \overline{W} - \alpha \sum_{i \in B} \frac{\partial L_i}{\partial \overline{W}}$$

- A batch B of training points is used in a single update of weights
- Increases the memory requirements. Layer outputs are matrices instead of vectors. In backward phase, matrices of gradients are calculated.
- Typical sizes are powers of 2 like 32, 64, 128, 256

Batch and Stochastic Gradient descent

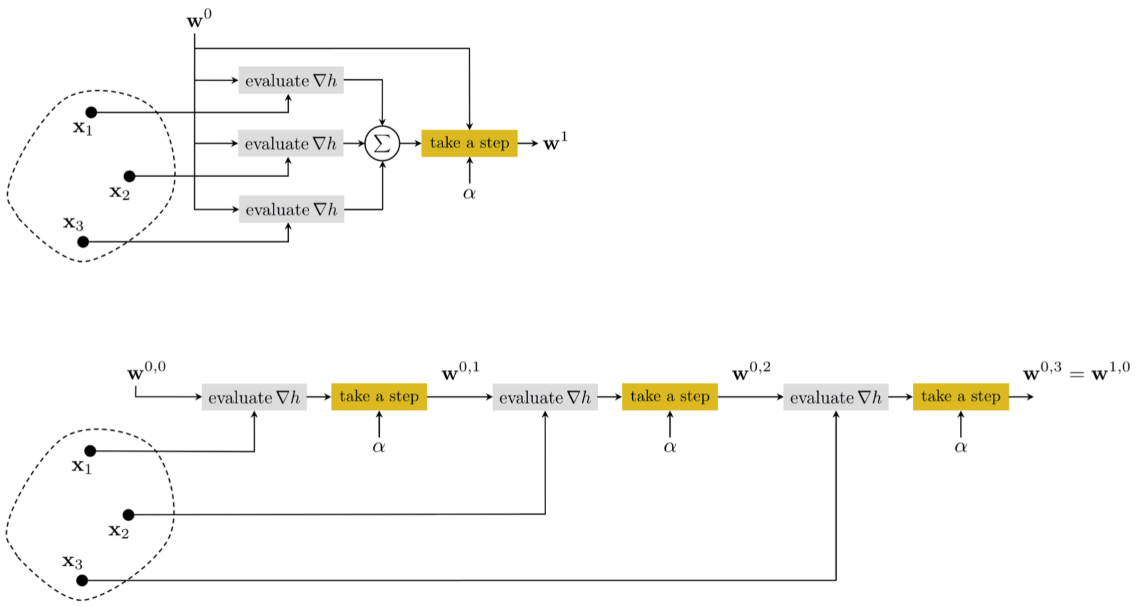
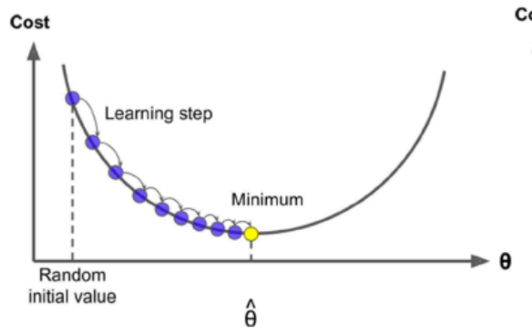


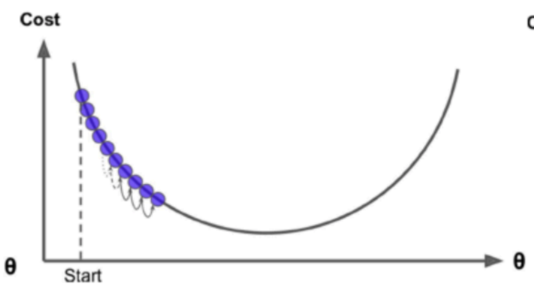
Figure 1: *Schematic comparison of first iteration of (top) batch and (bottom) stochastic gradient descent, using a dataset with three data points.*

https://kenndanielso.github.io/mlrefined/blog_posts/13_Multilayer_perceptrons/13_6_Stochastic_and_minibatch_gradient_descent.html

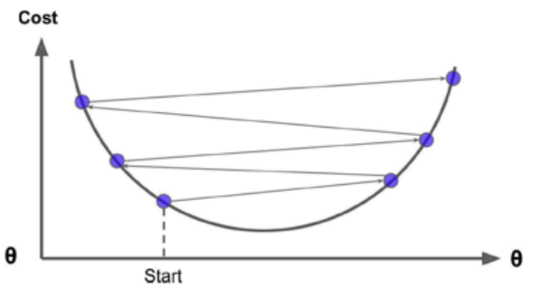
Learning rate: large value vs small value



<u>Optimum learning rate</u> : The model adjusts weights (θ) in subsequent training loops to arrive at cost minima.



<u>Slow learning rate</u> : Converges to cost minima but very slowly.



*<u>Fast learning rate</u> : **may** not converge to cost minima and the cost might keep increasing with further training loops.*

Image Credit : "Hands-on Machine Learning with Scikit-Learn and TensorFlow " by Aurelien Geron

Constant Learning Rate

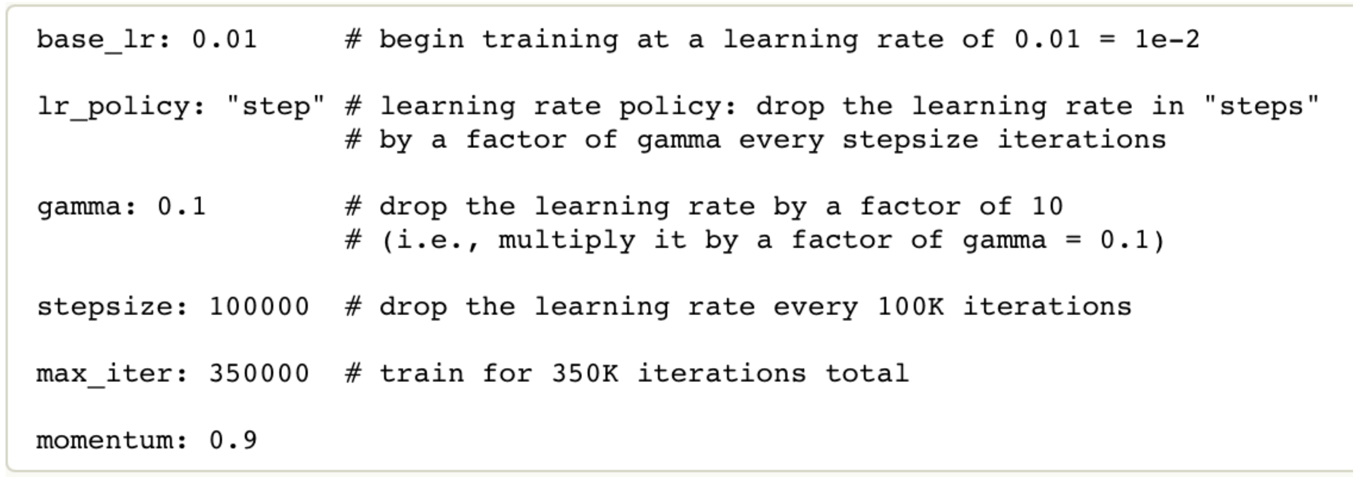
- Low learning rate may cause the algorithm to take too long time to come even close to an optimal solution
- Large learning rate will allow the algorithm to come close to a good solution but will then oscillate around the point or even diverge

Learning rate schedule

- Start with a higher learning rate to explore the loss space => find a good starting values for the weights
- Use smaller learning rates in later steps to converge to a minima =>tune the weights slowly
- Different choices of decay functions:
  - exponential, inverse, multi-step, polynomial
  - babysitting the learning rate
- Training with different learning rate decay
  - Keras learning rate schedules and decay
- Other new forms: cosine decay

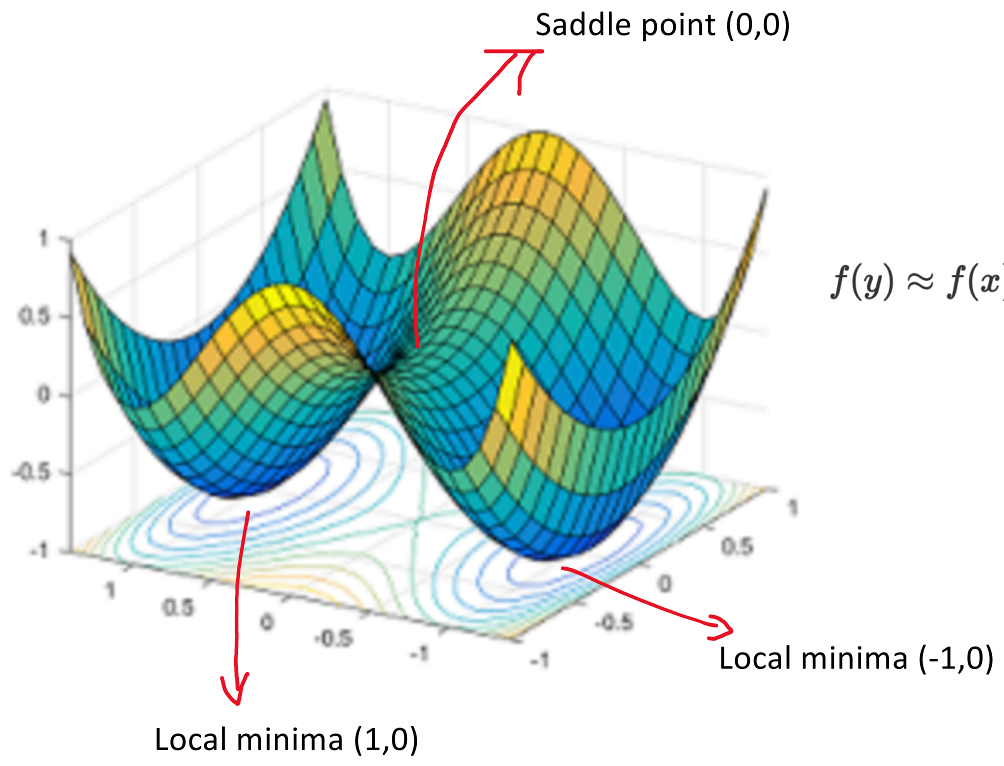
Decay functions	Decay equation
Inverse	$\alpha_t = \frac{\alpha_0}{1 + \gamma . t}$
exponential	$\alpha_t = \alpha_0 exp(-\gamma . t)$
polynomial n=1 gives linear	$\alpha_t = \alpha_0 \left(1 - \frac{t}{\max\_t}\right)^n$
multi-step	$\alpha_t = \frac{\alpha_0}{\gamma^n}$ at step n

Learning rate policy used in Alexnet



```
base_lr: 0.01     # begin training at a learning rate of 0.01 = 1e-2

lr_policy: "step" # learning rate policy: drop the learning rate in "steps"
                  # by a factor of gamma every stepsize iterations

gamma: 0.1        # drop the learning rate by a factor of 10
                  # (i.e., multiply it by a factor of gamma = 0.1)

stepsize: 100000  # drop the learning rate every 100K iterations

max_iter: 350000  # train for 350K iterations total

momentum: 0.9
```

Non-convex Loss Functions

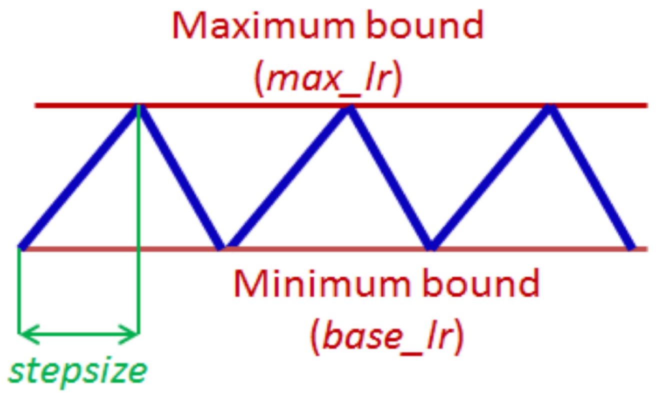


$$y = x_1^4 - 2x_1^2 + x_2^2$$

$$f(y) \approx f(x) + \langle \underline{\nabla f(x)}, y - x \rangle + \frac{1}{2}(y-x)^\top \nabla^2 f(x)(y-x)$$

https://www.offconvex.org/2016/03/22/saddlepoints/

Cyclical Learning Rate



Dataset	LR policy	Iterations	Accuracy (%)
CIFAR-10	*fixed*	70,000	81.4
CIFAR-10	*triangular2*	**25,000**	81.4
CIFAR-10	*decay*	25,000	78.5
CIFAR-10	*exp*	70,000	79.1
CIFAR-10	*exp_range*	42,000	**82.2**
AlexNet	*fixed*	400,000	58.0
AlexNet	*triangular2*	400,000	**58.4**
AlexNet	*exp*	300,000	56.0
AlexNet	*exp*	460,000	56.5
AlexNet	*exp_range*	300,000	56.5
GoogLeNet	*fixed*	420,000	63.0
GoogLeNet	*triangular2*	420,000	**64.4**
GoogLeNet	*exp*	240,000	58.2
GoogLeNet	*exp_range*	240,000	60.2

- Idea is to have learning rate continuously change in cyclical manner with alternate increase and decrease phases
- Keras implementation available; Look at example Cyclical Learning Rates with Keras and Deep Learning

Batch size

- Effect of batch size on learning
- Batch size is restricted by the GPU memory (12GB for K40, 16GB for P100 and V100) and the model size
    - Model and batch of data needs to remain in GPU memory for one iteration
- ResNet152 we need to stay below 10
- Are you doomed to work with small size mini-batches tfor large models and/or GPUs with limited memory
    - No, you can simulate large batch size by delaying gradient/weight updates to happen every n iterations (instead of n=1) ; supported by frameworks

Effective Mini-batch

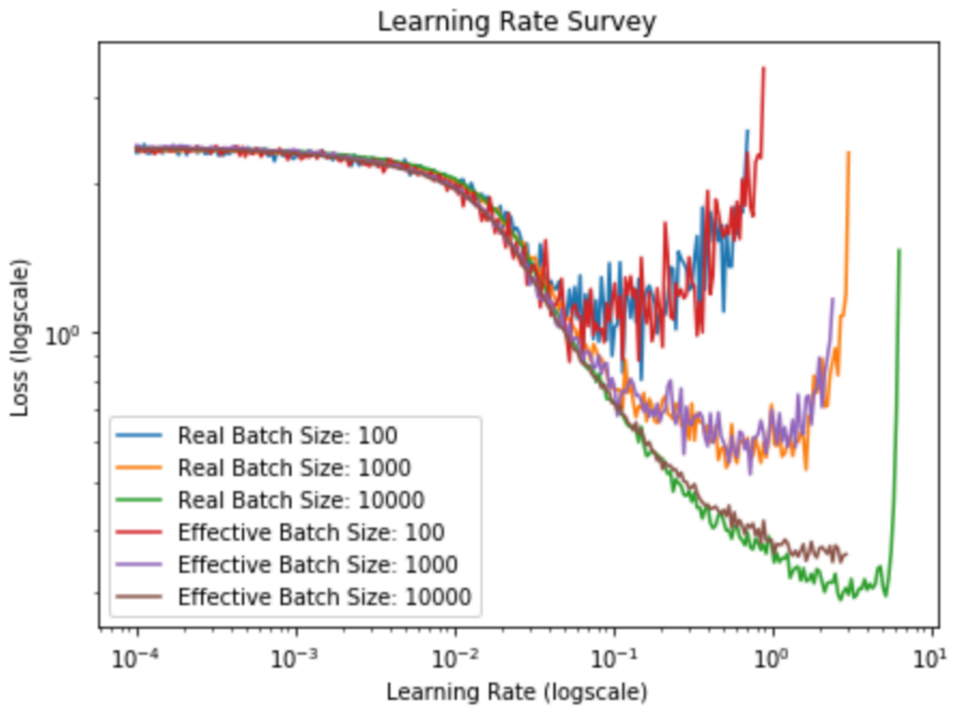
- Calculate and accumulate gradients over multiple mini-batches
- Perform optimizer step (update model parameters) only after specified number of mini-batches
- Caffe: iter_size; Pytorch: batch_multiplier

```
for inputs, targets in training_data_loader:
    optimizer.zero_grad()

    outputs = model(inputs)
    loss = loss_function(outputs, targets)
    loss.backward()

    optimizer.step()
```

```
count = 0
for inputs, targets in training_data_loader:
    if count == 0:
        optimizer.step()
        optimizer.zero_grad()
        count = batch_multiplier

    outputs = model(inputs)
    loss = loss_function(outputs, targets) / batch_multiplier
    loss.backward()

    count -= 1
```

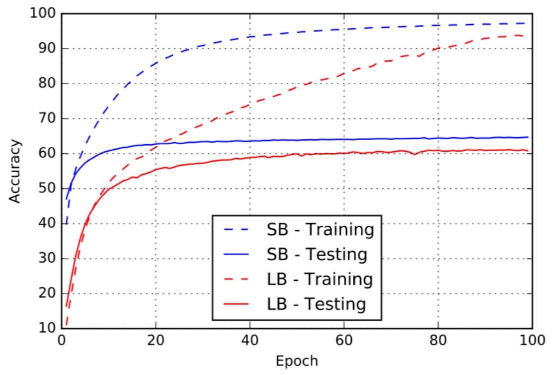
- Also remember to scale up the learning rate when working with large mini-batch size

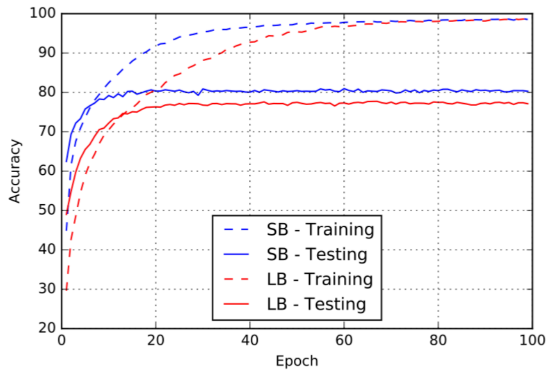
Effective Mini-batch Performance



What Batch size to choose ?

- Hardware constraints (GPU memory) dictate the largest batch size
- Should we try to work with the largest possible batch size ?
  - Large batch size gives more confidence in gradient estimation
  - Large batch size allows you to work with higher learning rates, faster convergence
- Large batch size leads to poor generalization (Keskar et al 2016)





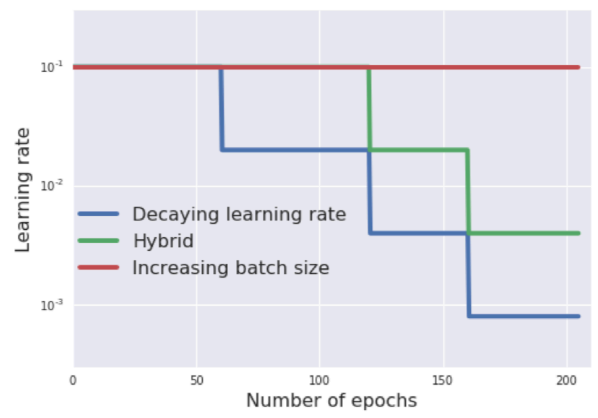
Learning rate and Batch size relationship

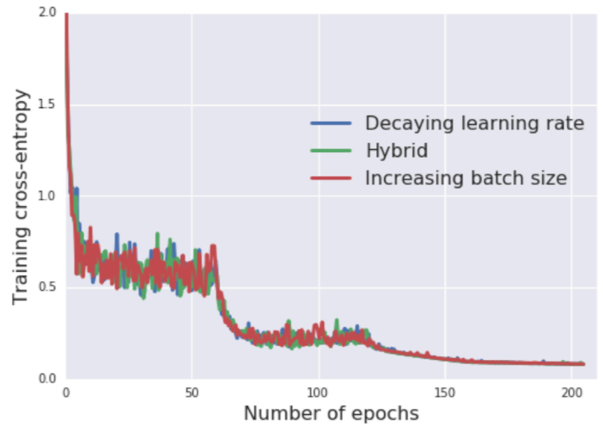
- "Noise scale" in stochastic gradient descent (Smith et al 2017)

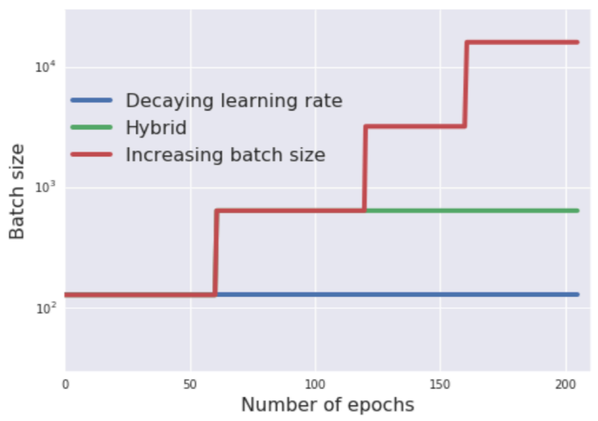
$g = \epsilon\left(\frac{N}{B} - 1\right)$	N: training dataset size
$g \approx \frac{\epsilon N}{B} \quad as\ B \ll N$	B: batch size
	ϵ: learning rate

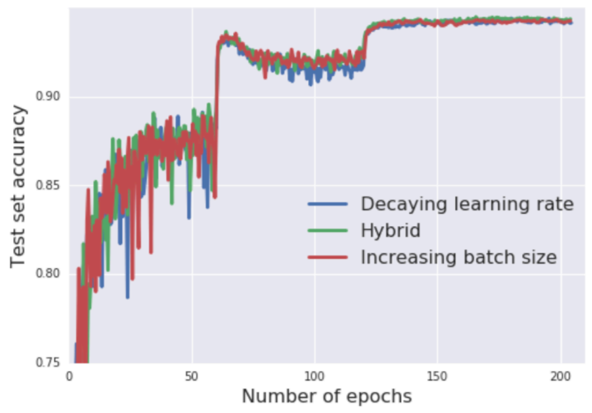
- There is an optimum fluctuation scale g which maximizes the test set accuracy (at constant learning rate)
  - Introduces an optimal batch size proportional to the learning rate when B $\ll$ N
- Increasing batch size will have the same effect as decreasing learning rate
  - Achieves near-identical model performance on the test set with the same number of training epochs but significantly fewer parameter updates

Learning rate decrease Vs Batch size increase









Prepare for Lecture 4

- Work on Reading-1 and Homework-1
- Seminar:
  - Form team of 2
  - Identify topic and associated papers
  - Should not be covered in class
  - Sign up for seminar slot
- Final Project:
  - Start thinking about project ideas, form team of 2 and submit your proposals for initial review/discussion
  - Project proposals due by Lecture 8
  - Proposals needs to be approved before you start working on it

Seminar Reading List

- **Batch Normalization**
  - Sergey Ioffe, Christian Szegedy Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift
  - Johan Bjorck, Carla Gomes, Bart Selman, Kilian Q. Weinberger Understanding Batch Normalization
  - Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, Aleksander Madry How Does Batch Normalization Help Optimization?
- **Learning rate and Batch size**
  - Samuel L. Smith, Pieter-Jan Kindermans, Chris Ying & Quoc V. Le DON'T DECAY THE LEARNING RATE, INCREASE THE BATCH SIZE
  - Keskar et al On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima
  - Leslie N. Smith Cyclical Learning Rates for Training Neural Networks
  - Elad Hoffer et al Augment your batch: better training with larger batches
- **Weight initialization**
  - Glorot and Bengio. Understanding the difficulty of training deep feedforward neural networks

Suggested Reading

- https://www.iro.umontreal.ca/~vincentp/ift3395/lectures/backprop_old.pdf Original paper on backpropagation
- Glorot et al Understanding the difficulty of training deep feedforward neural networks Paper introducing Glorot initialization
- Alex Sergeev Distributed Deep Learning (for lecture 4)
- Jeff Dean's ACM webinar on Deep Learning (for lecture 4)

Blogs/Code Links

- David Morton Increasing Mini-batch Size without Increasing Memory
- Adrian Rosebrock Keras learning rate schedules and decay