Neural Networks: Computation + Gradient Descent

LING572 Advanced Statistical Methods in NLP

February 27 2020

Today's Outline

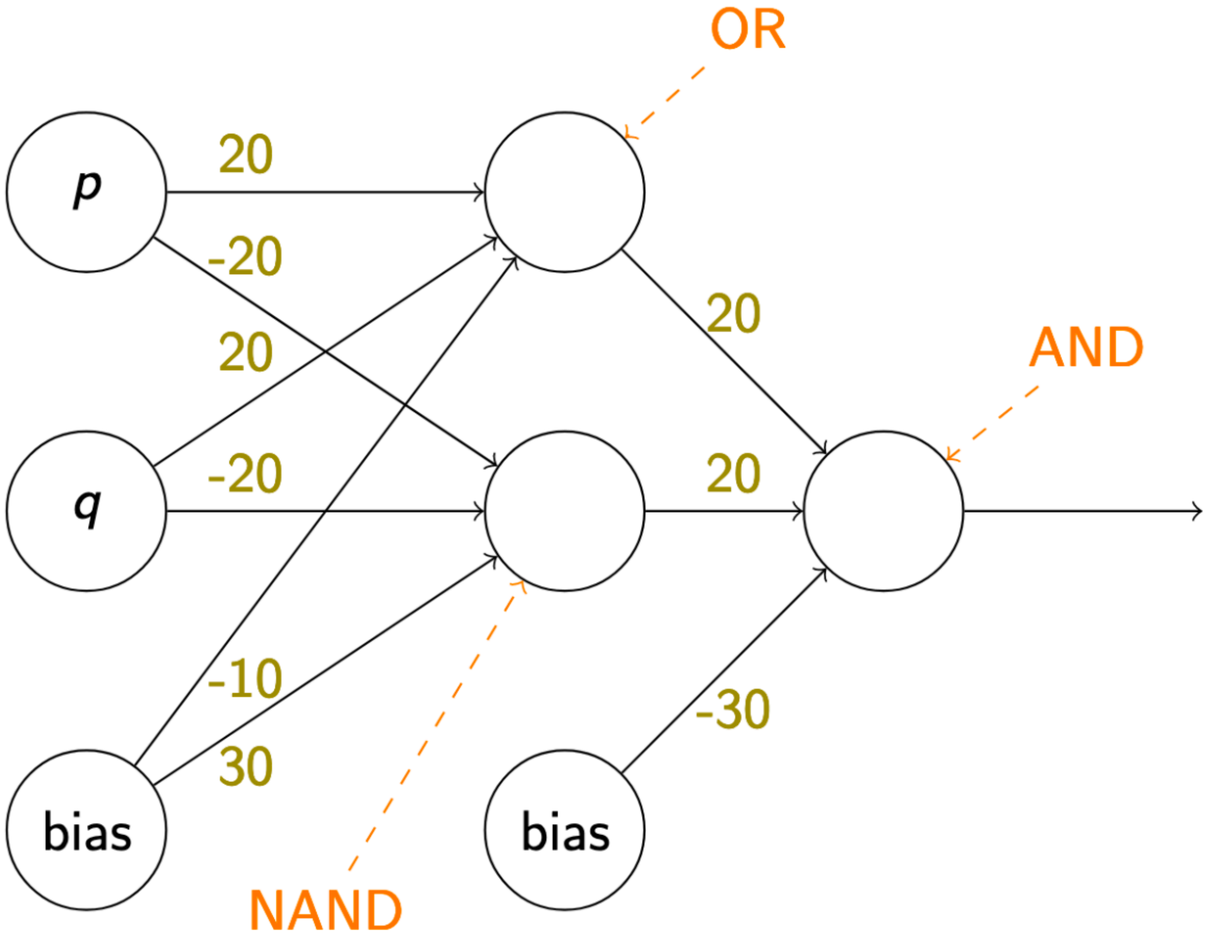
- Computation: the forward pass
  - Functional form / matrix notation
  - Parameters and Hyperparameters
- Gradient Descent
  - Intro
  - Stochastic Gradient Descent + Mini-batches

Notation

- I will generally use plain variables (e.g. x, y, W) for vectors and matrices as well as scalars, relying on context
- $\hat{y}$: a "guess" at y
  - e.g.: a *model's output*
- $f(x)$, when x is a vector/matrix means that f is applied *element-wise*
- θ*:* all parameters
- $\hat{y} = f(x; \theta) = f_\theta(x)$: $\hat{y}$ is a (parameterized) function of x with parameters θ

Feed-forward networks
aka Multi-layer perceptrons (MLP)

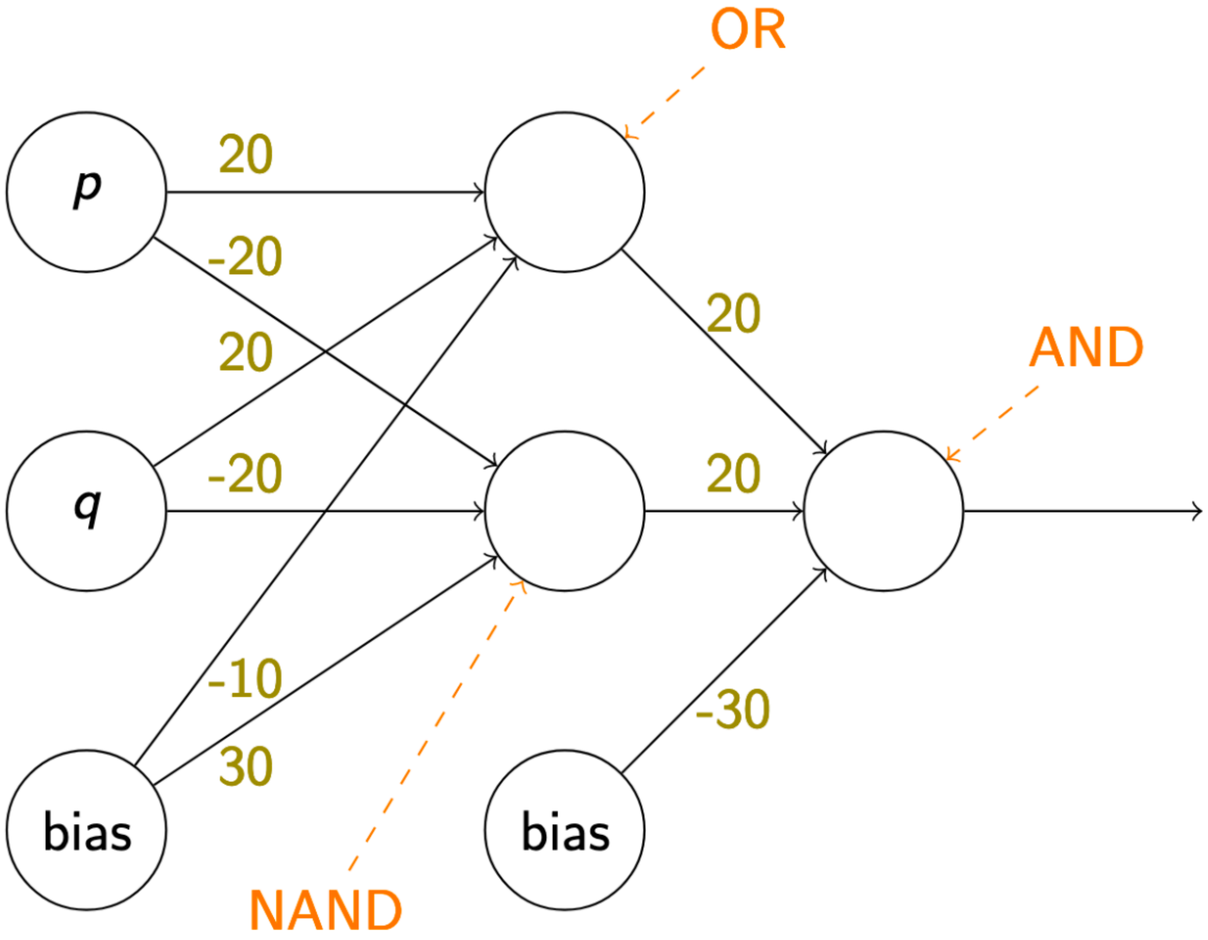
XOR Network



$$a_{\text{and}} = \sigma\left(w_{\text{or}}^{\text{and}} \cdot a_{\text{or}} + w_{\text{nand}}^{\text{and}} \cdot a_{\text{nand}} + b^{\text{and}}\right)$$

$$= \sigma\left(\begin{bmatrix} w_{\text{or}}^{\text{and}} & w_{\text{nand}}^{\text{and}} \end{bmatrix} \begin{bmatrix} a_{\text{or}} \\ a_{\text{nand}} \end{bmatrix} + b^{\text{and}}\right)$$

XOR Network



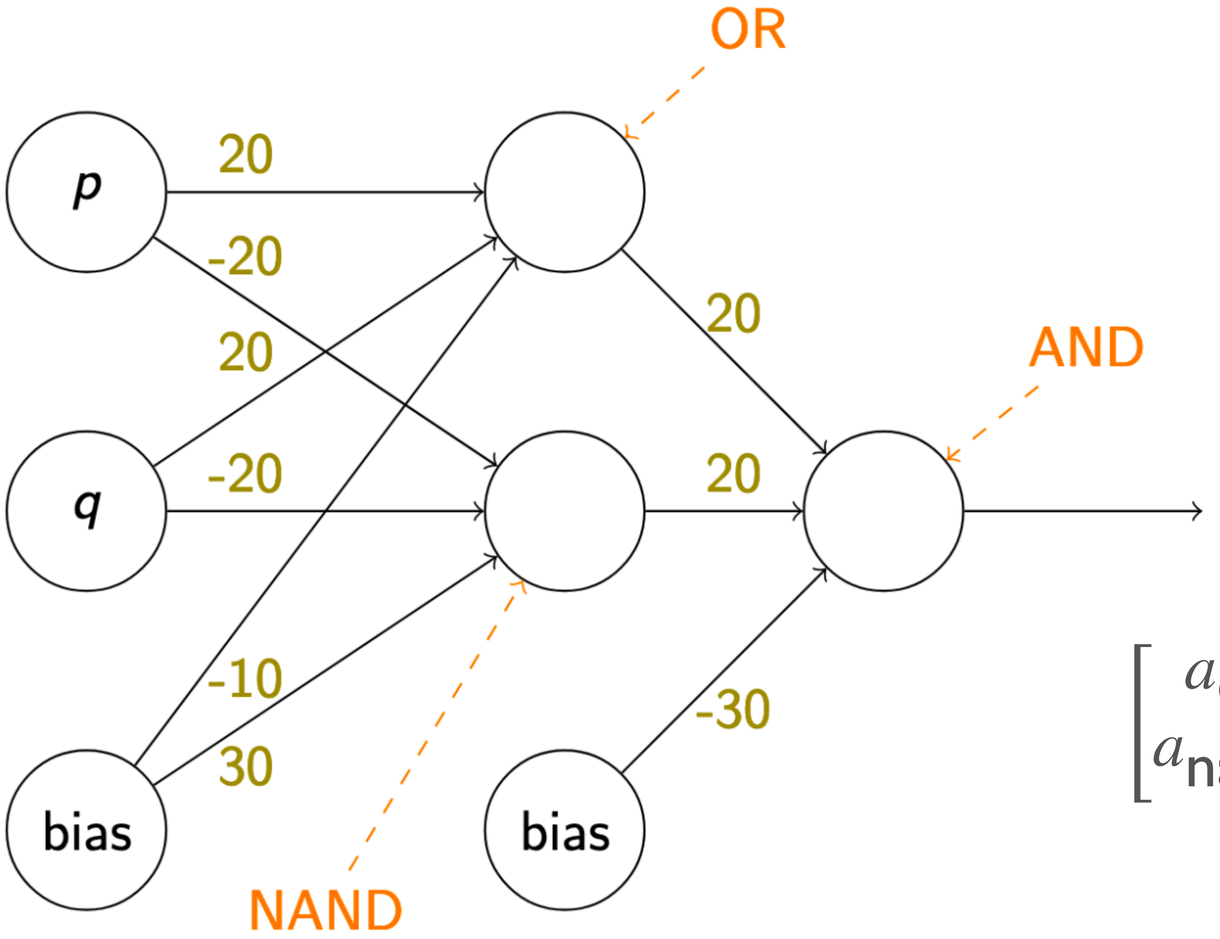
$$a_{\text{and}} = \sigma\left(w_{\text{or}}^{\text{and}} \cdot a_{\text{or}} + w_{\text{nand}}^{\text{and}} \cdot a_{\text{nand}} + b^{\text{and}}\right)$$

$$= \sigma\left(\begin{bmatrix} w_{\text{or}}^{\text{and}} & w_{\text{nand}}^{\text{and}} \end{bmatrix} \begin{bmatrix} a_{\text{or}} \\ a_{\text{nand}} \end{bmatrix} + b^{\text{and}}\right)$$

$$a_{\text{or}} = \sigma\left(w_p^{\text{or}} \cdot a_p + w_q^{\text{or}} \cdot a_q + b^{\text{or}}\right)$$

$$a_{\text{nand}} = \sigma\left(w_p^{\text{nand}} \cdot a_p + w_q^{\text{nand}} \cdot a_q + b^{\text{nand}}\right)$$

XOR Network

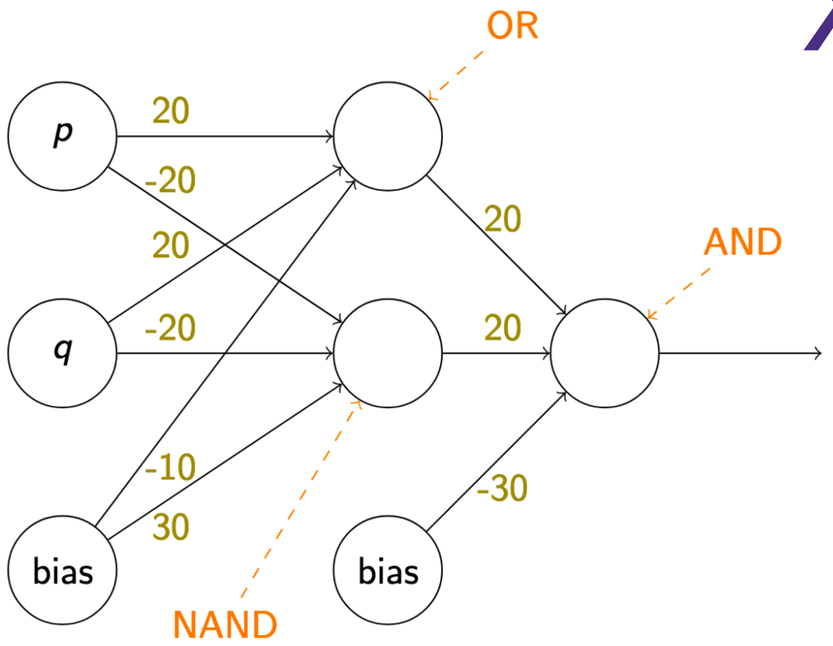


$$a_{\text{and}} = \sigma\left(w_{\text{or}}^{\text{and}} \cdot a_{\text{or}} + w_{\text{nand}}^{\text{and}} \cdot a_{\text{nand}} + b^{\text{and}}\right)$$

$$= \sigma\left(\begin{bmatrix} w_{\text{or}}^{\text{and}} & w_{\text{nand}}^{\text{and}} \end{bmatrix} \begin{bmatrix} a_{\text{or}} \\ a_{\text{nand}} \end{bmatrix} + b^{\text{and}}\right)$$

$$\begin{bmatrix} a_{\text{or}} \\ a_{\text{nand}} \end{bmatrix} = \sigma\left(\begin{bmatrix} w_p^{\text{or}} & w_q^{\text{or}} \\ w_p^{\text{nand}} & w_q^{\text{nand}} \end{bmatrix} \begin{bmatrix} a_p \\ a_q \end{bmatrix} + \begin{bmatrix} b^{\text{or}} \\ b^{\text{nand}} \end{bmatrix}\right)$$

XOR Network



$$a_{\text{and}} = \sigma\left(w_{\text{or}}^{\text{and}} \cdot a_{\text{or}} + w_{\text{nand}}^{\text{and}} \cdot a_{\text{nand}} + b^{\text{and}}\right)$$

$$= \sigma\left(\begin{bmatrix} w_{\text{or}}^{\text{and}} & w_{\text{nand}}^{\text{and}} \end{bmatrix} \begin{bmatrix} a_{\text{or}} \\ a_{\text{nand}} \end{bmatrix} + b^{\text{and}}\right)$$

$$a_{\text{and}} = \sigma\left(\begin{bmatrix} w_{\text{or}}^{\text{and}} & w_{\text{nand}}^{\text{and}} \end{bmatrix} \sigma\left(\begin{bmatrix} w_p^{\text{or}} & w_q^{\text{or}} \\ w_p^{\text{nand}} & w_q^{\text{nand}} \end{bmatrix} \begin{bmatrix} a_p \\ a_q \end{bmatrix} + \begin{bmatrix} b^{\text{or}} \\ b^{\text{nand}} \end{bmatrix}\right) + b^{\text{and}}\right)$$

Generalizing

$$a_{\text{and}} = \sigma\left(\begin{bmatrix} w_{\text{or}}^{\text{and}} & w_{\text{nand}}^{\text{and}} \end{bmatrix} \sigma\left(\begin{bmatrix} w_p^{\text{or}} & w_q^{\text{or}} \\ w_p^{\text{nand}} & w_q^{\text{nand}} \end{bmatrix} \begin{bmatrix} a_p \\ a_q \end{bmatrix} + \begin{bmatrix} b^{\text{or}} \\ b^{\text{nand}} \end{bmatrix}\right) + b^{\text{and}}\right)$$

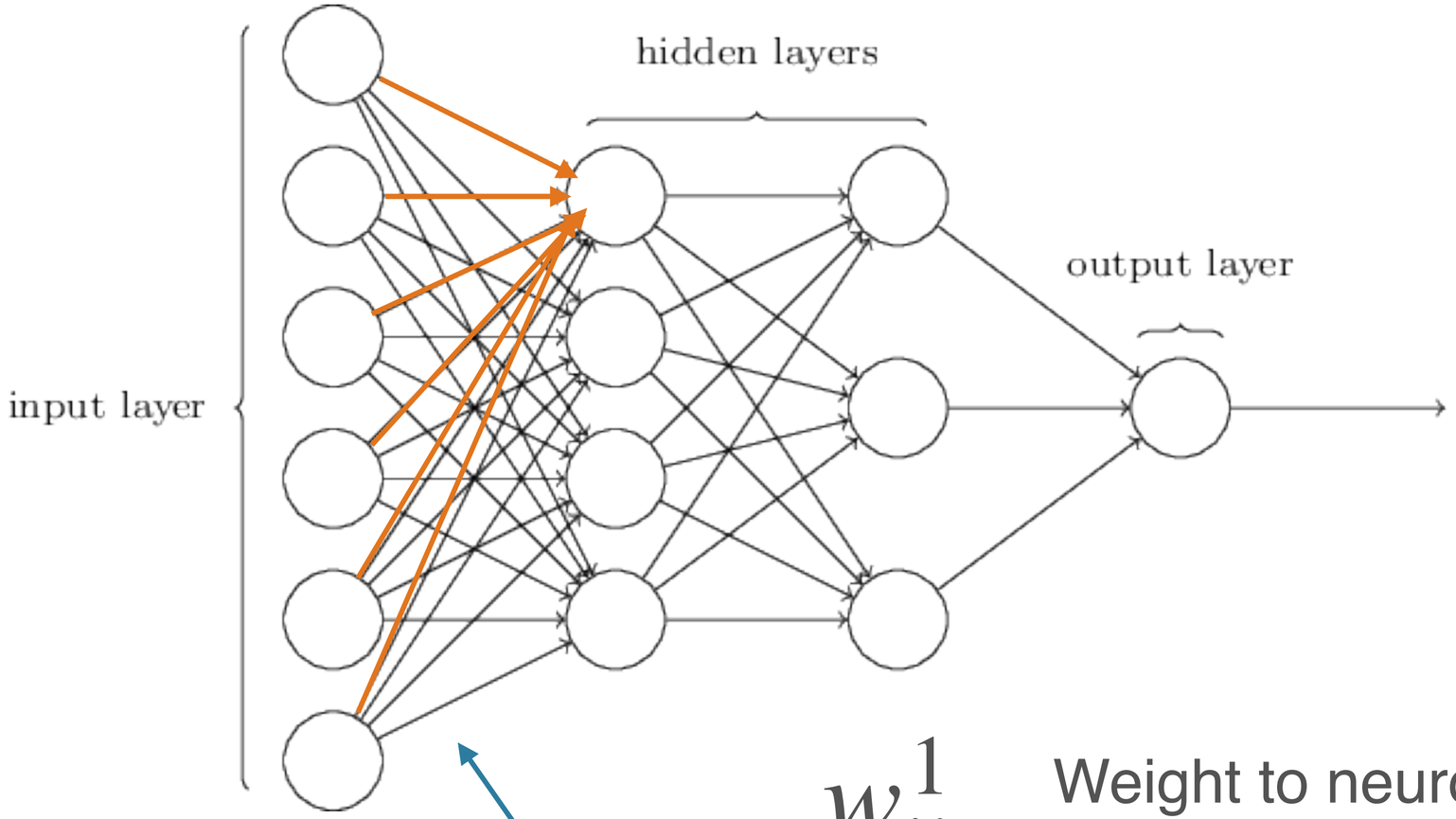
$$\hat{y} = f_2\left(W^2 f_1\left(W^1 x + b^1\right) + b^2\right)$$

$$\hat{y} = f_n\left(W^n f_{n-1}\left(\cdots f_2\left(W^2 f_1\left(W^1 x + b^1\right) + b^2\right)\cdots\right) + b^n\right)$$

Some terminology

- Our XOR network is a *feed-forward neural network* with *one hidden layer*
  - Aka a multi-layer perceptron (MLP)
- Input nodes: 2; output nodes: 1
- Activation function: sigmoid

General MLP



source

W^1

w_{ij}^1

Weight to neuron i in layer 1
from neuron j in layer 0

General MLP

$$\hat{y} = f_n\left(W^n f_{n-1}\left(\cdots f_2\left(W^2 f_1\left(W^1 x + b^1\right) + b^2\right)\cdots\right) + b^n\right)$$

$$W^1 = \begin{bmatrix} w^1_{00} & w^1_{01} & \cdots & w^1_{0n_0} \\ w^1_{10} & w^1_{11} & \cdots & w^1_{1n_0} \\ \vdots & \vdots & \ddots & \vdots \\ w^1_{n_1 0} & w^1_{n_1 1} & \cdots & w^1_{n_1 n_0} \end{bmatrix}$$

Shape: (n_1, n_0)

$$x = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n_0} \end{bmatrix}$$

Shape: $(n_0, 1)$

$$b^1 = \begin{bmatrix} b^1_0 \\ b^1_1 \\ \vdots \\ b^1_{n_1} \end{bmatrix}$$

Shape: $(n_1, 1)$

n_0: number of neurons in layer 0 (input)

n_1: number of neurons in layer 1

Parameters of an MLP

- Weights and biases
  - For each layer l: $n_l(n_{l-1} + 1)$
  - $n_l n_{l-1}$ weights; n_l biases
- With *n* hidden layers (considering the output as a hidden layer):

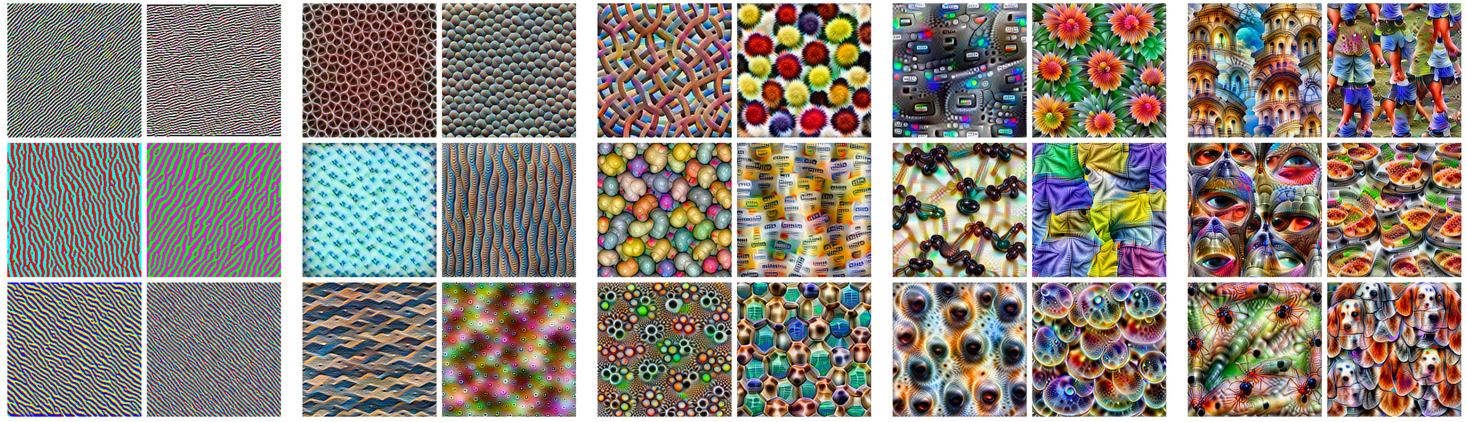
$$\sum_{i=1}^{n} n_i(n_{i-1} + 1)$$

Hyper-parameters of an MLP

- Input size, output size
  - Usually fixed by your problem / dataset
  - Input: image size, vocab size; number of “raw” features in general
  - Output: 1 for binary classification or simple regression, number of labels for classification, …
- *Number* of hidden layers
- For each hidden layer:
  - Size
  - Activation function
- Others: initialization, regularization (and associated values), learning rate / training, …

The Deep in Deep Learning

- The Universal Approximation Theorem says that one hidden layer suffices for arbitrarily-closely approximating a given function
- Empirical drawbacks: Super-exponentially many neurons; hard to discover
- “Deep and narrow” >> “Shallow and wide”
  - In principle allows hierarchical features to be learned
  - More well-behaved w/r/t optimization



Edges (layer conv2d0) **Textures** (layer mixed3a) **Patterns** (layer mixed4a) **Parts** (layers mixed4b & mixed4c) **Objects** (layers mixed4d & mixed4e)

source

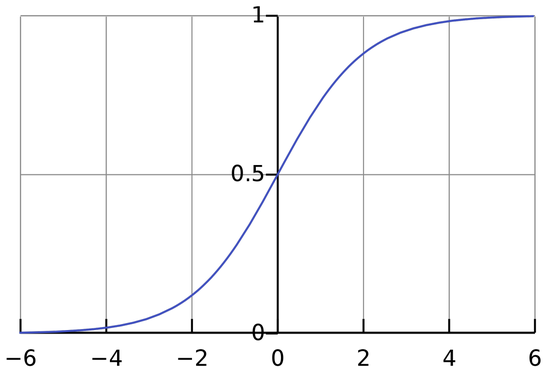
Activation Functions

- Note: *non-linear* activation functions are essential
- MLP: linear transformation, followed by a point-wise non-linearity, repeated several times over
- Without the non-linearity, would just have several linear transformations
  - Composition of linear transformations is *also* linear!

$$\hat{y} = f_n\left(W^n f_{n-1}\left(\cdots f_2\left(W^2 f_1\left(W^1 x + b^1\right) + b^2\right)\cdots\right) + b^n\right)$$

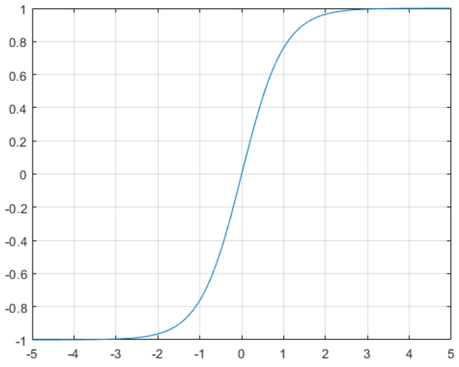
Activation Functions: Hidden Layer

sigmoid

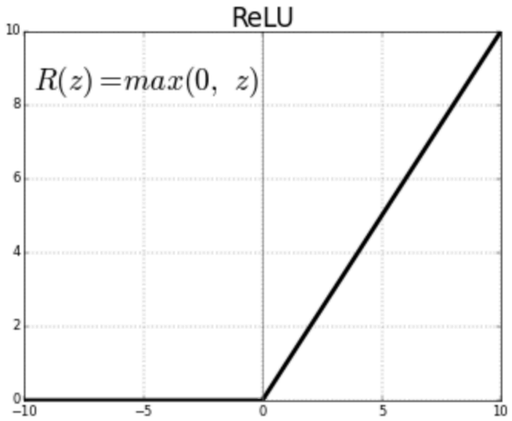


$$\sigma(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1}$$

tanh



$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\sigma(2x) - 1$$



- Use ReLU by default
- Generalizations:
  - Leaky
  - ELU
  - Softplus
  - …

Problem: derivative “saturates” (nearly 0) everywhere except near origin

Activation Functions: Output Layer

- Depends on the task!
- Regression (continuous output(s)): none!
  - Just use final linear transformation
- Binary classification: sigmoid
  - Also for *multi-label* classification
- Multi-class classification: softmax
  - Terminology: the inputs to a softmax are called *logits*
  - [there are sometimes other uses of the term, so beware]

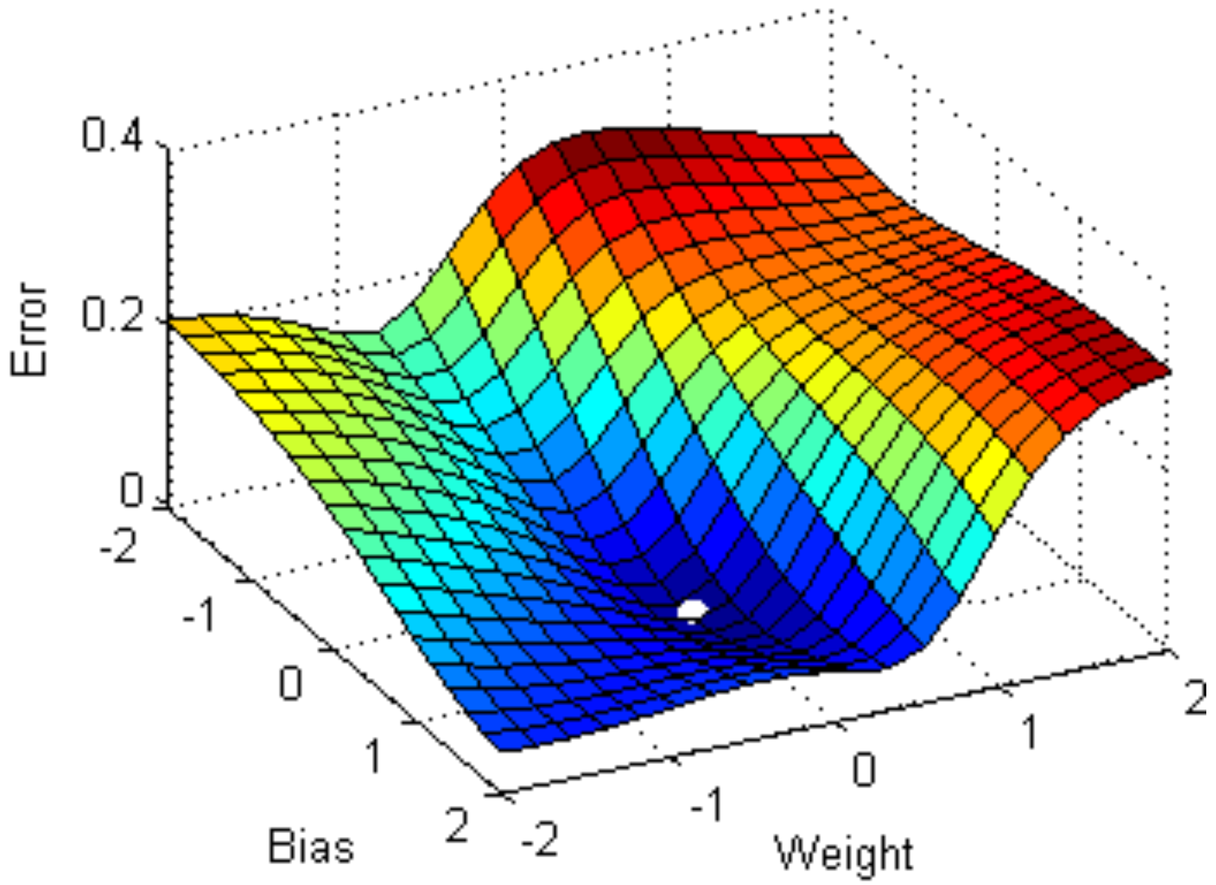
$$\text{softmax}(x)_i = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Learning: (Stochastic) Gradient Descent

Gradient Descent: Basic Idea

- Treat NN training as an optimization problem
- $\ell(\hat{y}, y)$: loss function ("objective function"); $\mathcal{L}(\hat{Y}, Y) = \frac{1}{|Y|} \sum_i \ell(\hat{y}(x_i), y_i)$
  - How "close" is the model's output to the true output
  - Local loss, averaged over training instances
  - More later: depends on the particular task, among other things
- View the loss *as a function of the model's parameters*
- The *gradient* of the loss w/r/t parameters tells which direction in parameter space to "walk" to make the loss smaller (i.e. to improve model outputs)
- Guaranteed to work in linear case; can get stuck in local minima for NNs

Gradient Descent: Basic Idea



source

Derivatives

- The derivative of a function of one real variable measures how much the output changes with respect to a change in the input variable

$$f(x) = x^2 + 35x + 12$$

$$\frac{df}{dx} = 2x + 35$$

$$f(x) = e^x$$

$$\frac{df}{dx} = e^x$$

Partial Derivatives

- A partial derivative of a function of several variables measures its derivative with respect one of those variables, with the others held constant.

$$f(x) = 10x^3y^2 + 5xy^3 + 4x + y$$

$$\frac{\partial f}{\partial x} = 30x^2y^2 + 5y^3 + 4$$

$$\frac{\partial f}{\partial y} = 20x^3y + 15xy^2 + 1$$

Gradient

- The gradient of a function $f(x_1, x_2, \ldots x_n)$ is a vector function, returning all of the partial derivatives

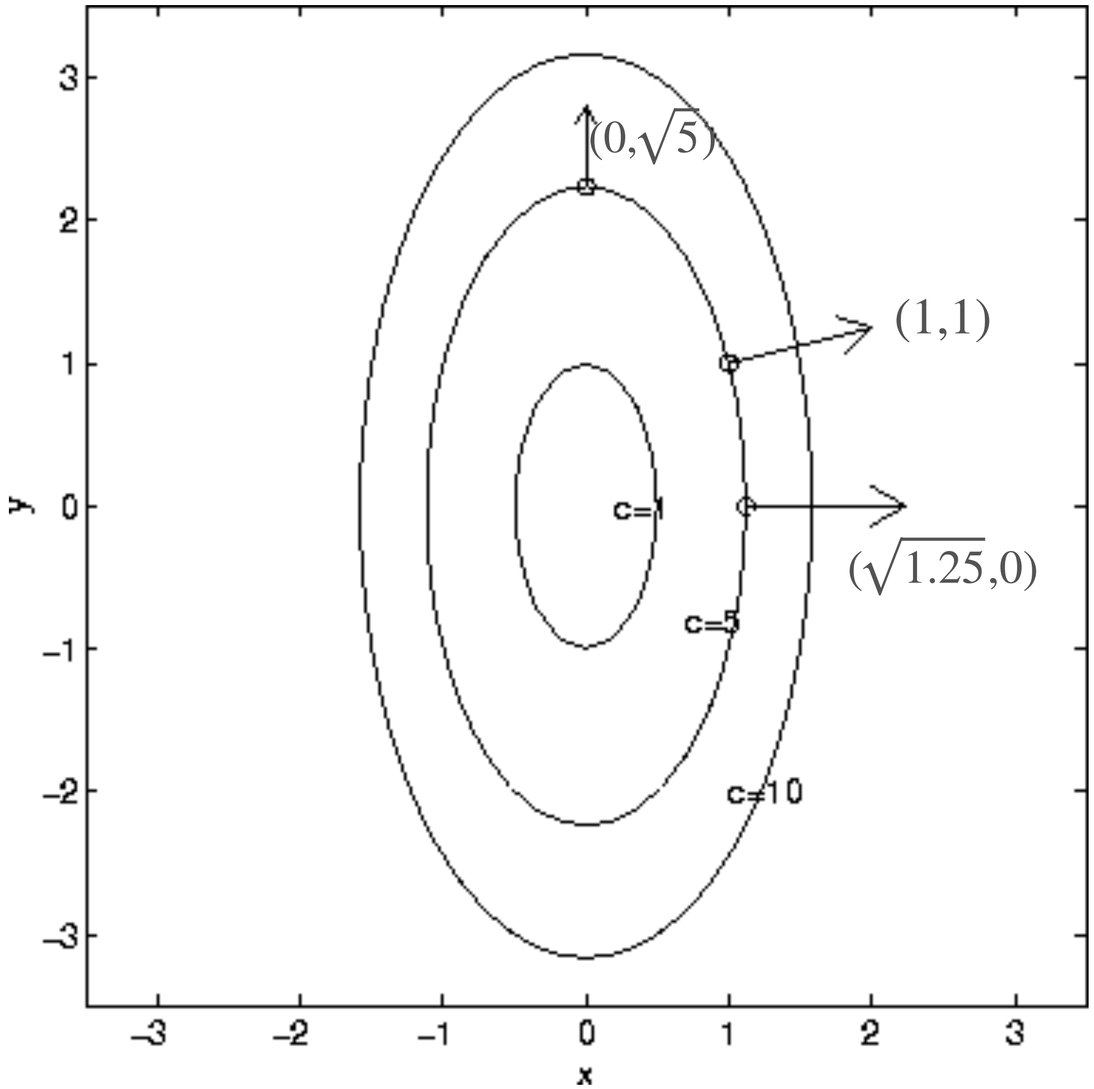
$$\nabla f = \left\langle \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right\rangle$$

$$f(x) = 4x^2 + y^2$$

$$\nabla f = \langle 8x, 2y \rangle$$

- The gradient is perpendicular to the *level curve* at a point
- The gradient points in the direction of greatest rate of increase of f

Gradient and Level Curves



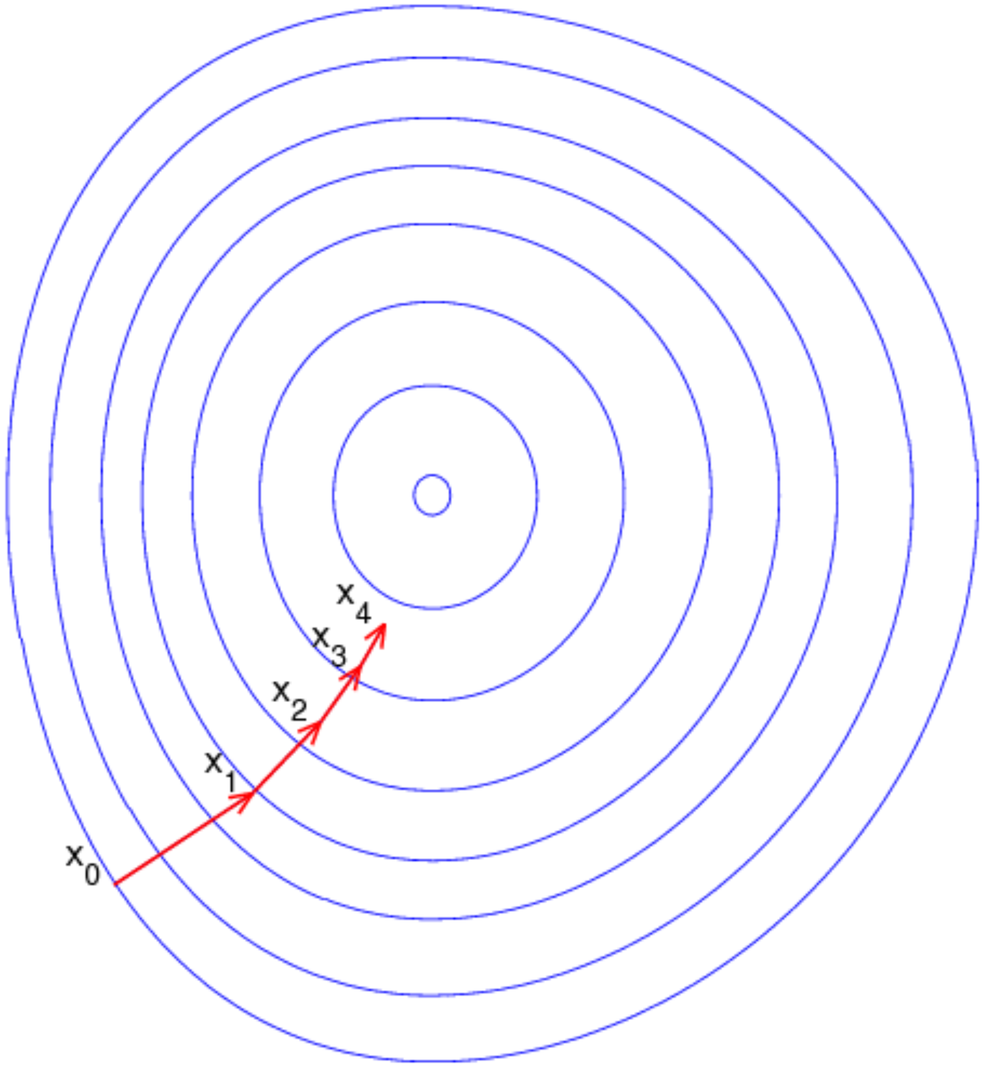
$$f(x) = 4x^2 + y^2$$

$$\nabla f = \langle 8x, 2y \rangle$$

Level curves: $f(x) = c$

Q: what are the actual gradients at those points?

Gradient Descent and Level Curves



source

Gradient Descent Algorithm

- Initialize θ_0
- Repeat until convergence:

$$\theta_{n+1} = \theta_n - \alpha \nabla \mathscr{L}(\hat{Y}(\theta_n), Y)$$

Learning rate

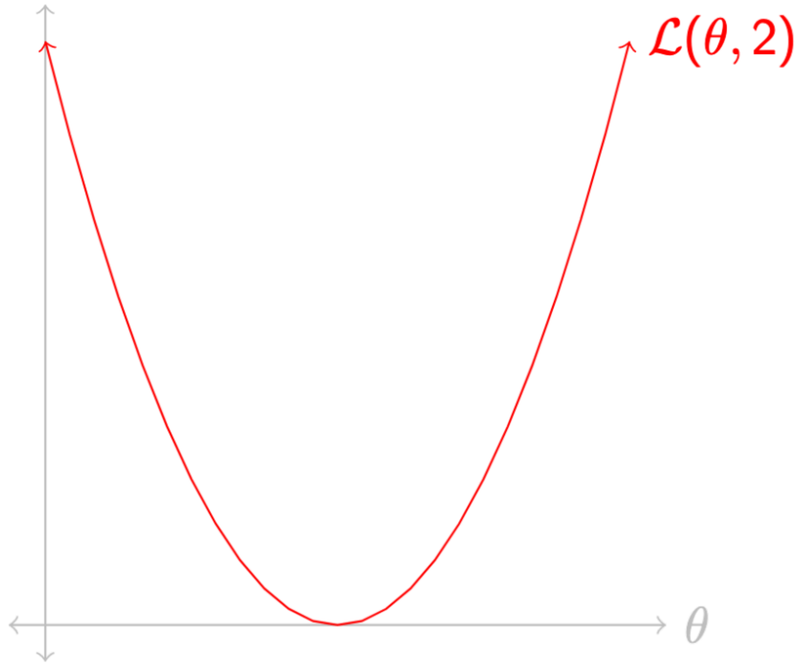
- High learning rate: big steps, may bounce and “overshoot” the target
- Low learning rate: small steps, smoother minimization of loss, but can be slow

Gradient Descent: Minimal Example

- Task: predict a target/true value $y = 2$
- "Model": $\hat{y}(\theta) = \theta$
  - A single parameter: the actual guess
- Loss: Euclidean distance

$$\mathscr{L}(\hat{y}(\theta), y) = (\hat{y} - y)^2 = (\theta - y)^2$$

Gradient Descent: Minimal Example



$$\frac{\partial}{\partial \theta}\mathcal{L}(\theta, y) = 2(\theta - y)$$

$$\theta_{t+1} = \theta_t - \alpha \cdot \frac{\partial}{\partial \theta}\mathcal{L}(\theta, y)$$

Stochastic Gradient Descent

- The above is called “batch” gradient descent
  - Updates *once per pass through the dataset*
  - Expensive, and slow; does not scale well
- *Stochastic* gradient descent:
  - Break the data into “mini-batches”: small chunks of the data
  - Compute gradients and update parameters for each batch
  - Mini-batch of size 1 = single example
  - A *noisy estimate* of the true gradient, but works well in practice; more parameter updates
- Epoch: one pass through the whole training data

Stochastic Gradient Descent

```
initialize parameters / build model

for each epoch:

  data = shuffle(data)
  batches = make_batches(data)

  for each batch in batches:

    outputs = model(batch)
    loss = loss_fn(outputs, true_outputs)
    compute gradients // e.g. loss.backward()
    update parameters
```

Computing with Mini-batches

- Bad idea:

```
for each batch in batches:
  for each datum in batch:
    outputs = model(datum)
    loss = loss_fn(outputs, true_outputs)
    compute gradients // e.g. loss.backward()
    update parameters
```

Computing with a Single Input

$$\hat{y} = f_n\left(W^n f_{n-1}\left(\cdots f_2\left(W^2 f_1\left(W^1 x + b^1\right) + b^2\right)\cdots\right) + b^n\right)$$

$$W^1 = \begin{bmatrix} w^1_{00} & w^1_{01} & \cdots & w^1_{0n_0} \\ w^1_{10} & w^1_{11} & \cdots & w^1_{1n_0} \\ \vdots & \vdots & \ddots & \vdots \\ w^1_{n_1 0} & w^1_{n_1 1} & \cdots & w^1_{n_1 n_0} \end{bmatrix}$$

Shape: (n_1, n_0)

n_0: number of neurons in layer 0 (input)

n_1: number of neurons in layer 1

$$x = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n_0} \end{bmatrix}$$

Shape: $(n_0, 1)$

$$b^1 = \begin{bmatrix} b^1_0 \\ b^1_1 \\ \vdots \\ b^1_{n_1} \end{bmatrix}$$

Shape: $(n_1, 1)$

Computing with a Batch of Inputs

$$\hat{y} = f_n\left(f_{n-1}\left(\cdots f_2\left(f_1\left(xW^1 + b^1\right)W^2 + b^2\right)\cdots\right)W^n + b^n\right)$$

$$x = \begin{bmatrix} x_0^0 & x_1^0 & \dots & x_{n_0}^0 \\ x_1^0 & x_1^1 & \dots & x_{n_0}^1 \\ \vdots & \vdots & \ddots & \vdots \\ x_1^n & x_1^n & \dots & x_{n_0}^n \end{bmatrix}$$

Shape: (n, n_0)
n: batch_size

$$W^1 = \begin{bmatrix} w_{00}^1 & w_{01}^1 & \cdots & w_{0n_1}^1 \\ w_{10}^1 & w_{11}^1 & \cdots & w_{1n_1}^1 \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_0 0}^1 & w_{n_0 1}^1 & \cdots & w_{n_0 n_1}^1 \end{bmatrix}$$

Shape: (n_0, n_1)
n_0: number of neurons in layer 0 (input)
n_1: number of neurons in layer 1

$$b^1 = \begin{bmatrix} b_0^1 & b_1^1 & \dots & b_{n_1}^1 \end{bmatrix}$$

Shape: $(1,n_1)$
Added to each row of xW^1

Note on mini-batches and shape

- Most modern neural net libraries (e.g. PyTorch) expect the *first* dimension of matrices/tensors to be a batch size
  - Produce a sequence of representations, *for each item* in the batch
  - e.g. (batch_size, input_size) —> (batch_size, hidden_size) —> (batch_size, output_size)
- In principle, can be higher than 2-dimensional
  - Images: (batch_size, width, height, 3)
  - Sequences: (batch_size, seq_len, representation_size)
- Two comments:
  - In your code, **annotate every tensor** with a comment saying intended shape
  - When debugging, look at shapes early on!!

Regularization

- NNs are often *overparameterized*, so regularization helps
- L1/L2: $\mathscr{L}'(\theta, y) = \mathscr{L}(\theta, y) + \lambda \|\theta\|^2$
- Dropout (2012):
  - *During training*, randomly turn off X% of neurons in each layer
  - (Don't do this during testing/predicting)
- Batch Normalization (2015)

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

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

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

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

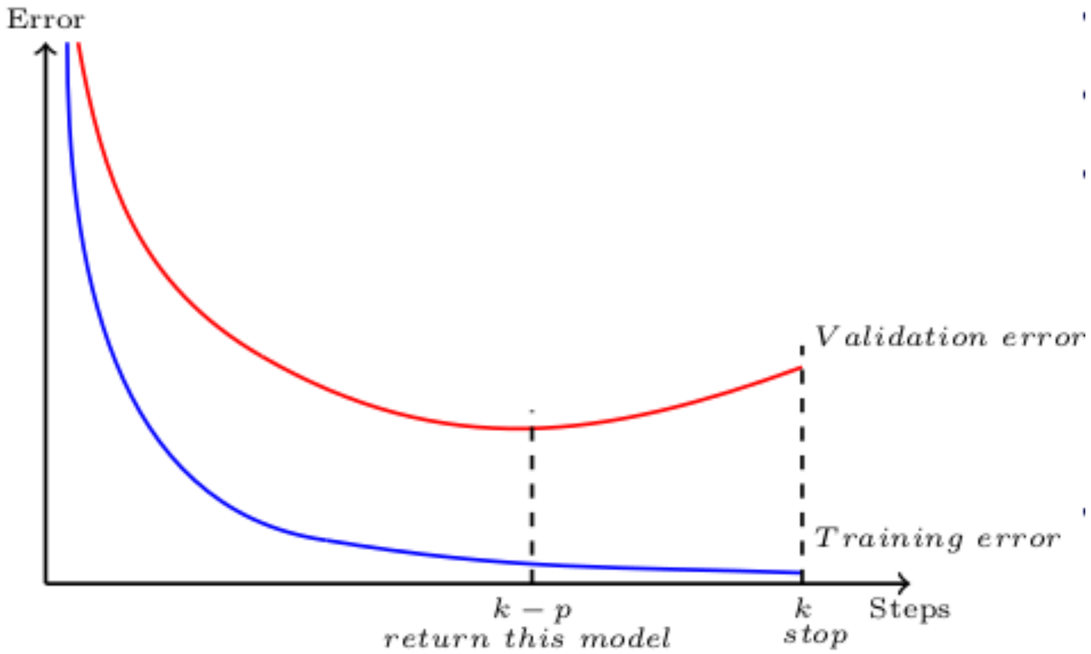
$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Hyper-parameters

- In addition to the model architecture ones mentioned earlier
- Optimizer: SGD, Adam, Adagrad, RMSProp, ….
  - Optimizer-specific hyper-parameters: learning rate, alpha, beta, …
  - NB: backprop computes gradients; optimizer uses them to update parameters
- Regularization: L1/L2, Dropout, BN, …
  - regularizer-specific ones: e.g. dropout rate
- Batch size
- Number of epochs to train for
  - Early stopping criterion (e.g. number of epochs, “patience”)

Early stopping

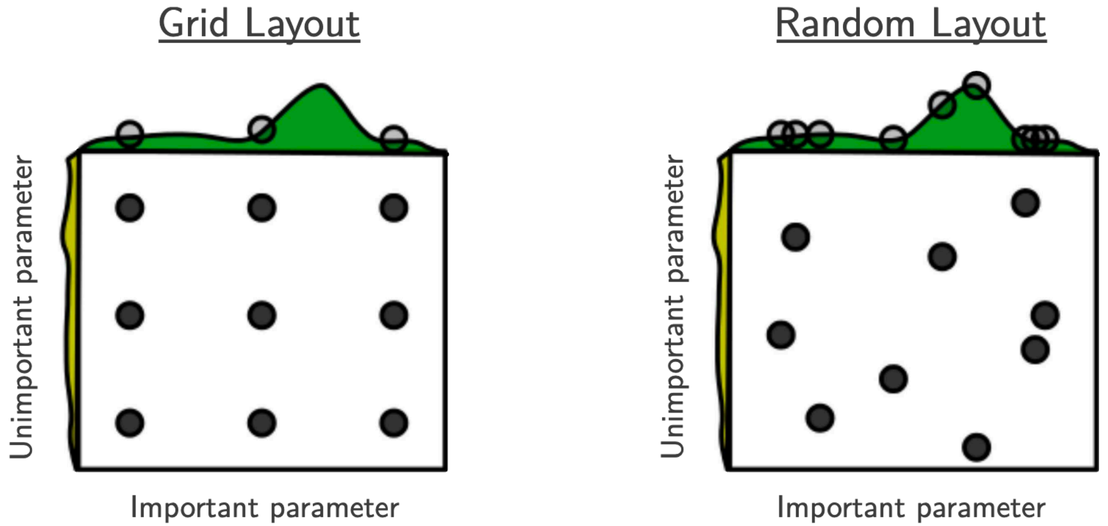
- One: Pick # of epochs, hope for no overfitting
- Better: pick max # of epochs, and “patience”
  - Halt when validation error does not improve over patience-many epochs



source

A note on hyper-parameter tuning

- Grid search: specify range of values for each hyper-parameter, try all possible combinations thereof
- Random search: specify possible values for all parameters, randomly sample values for each, stop when some criterion is met



Bergstra and Bengio 2012

Next time

- Today: how to train an NN by SGD
  - Compute gradients of loss w/r/t parameters
  - Update parameters (weights) in the opposite direction, to minimize loss
- Next time:
  - How do we compute gradients???
  - Backpropagation