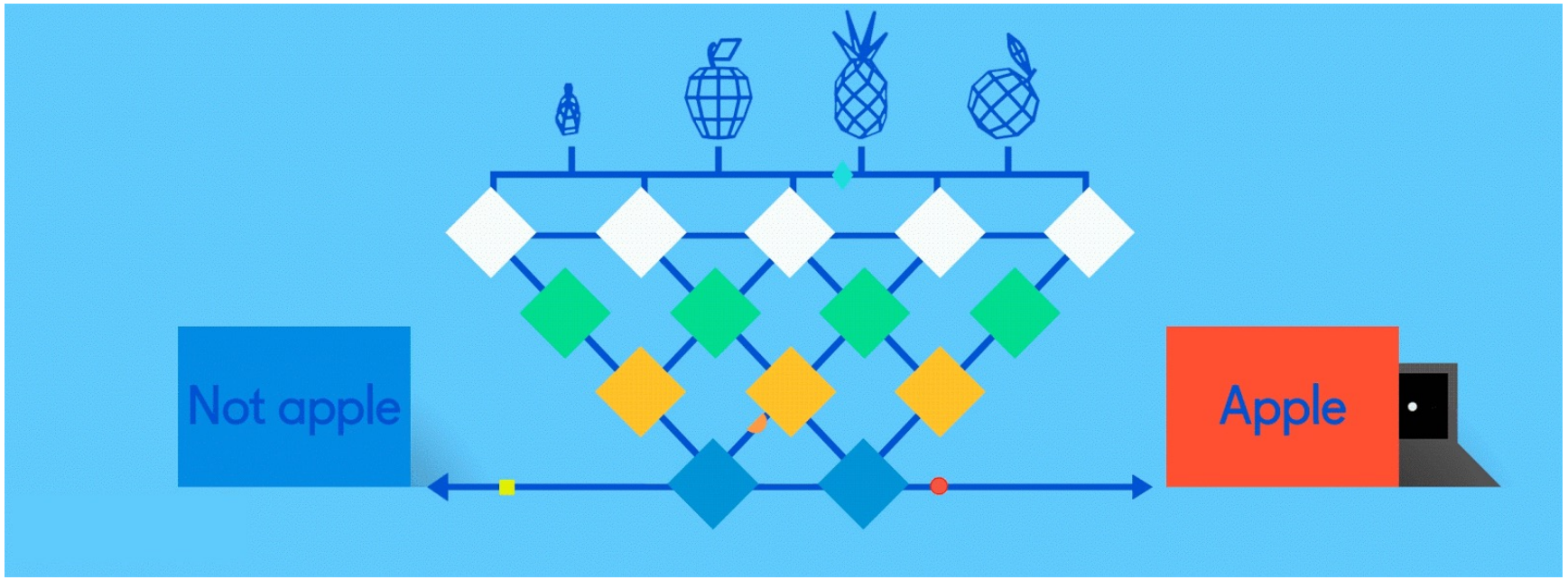
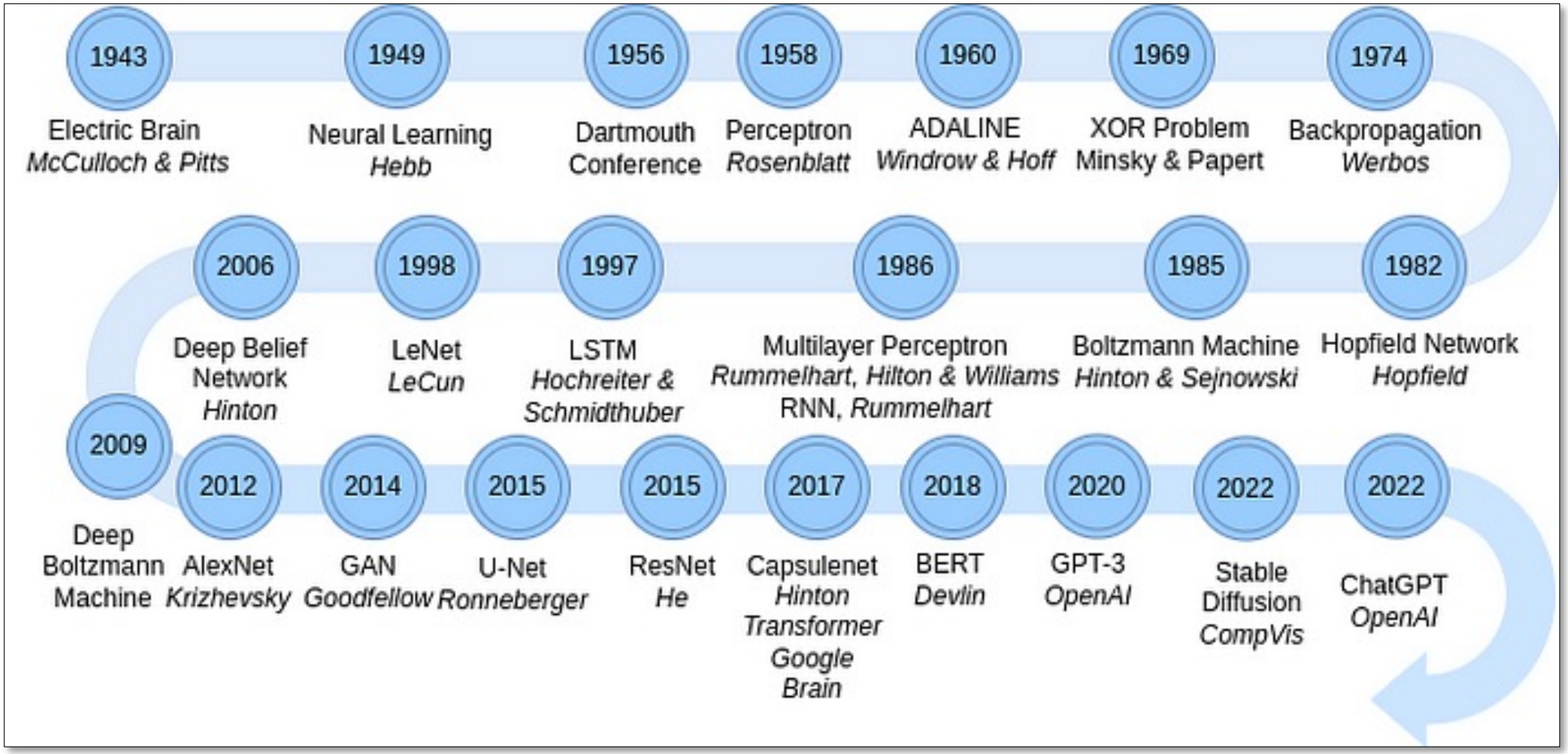
Neural Networks for Machine Learning

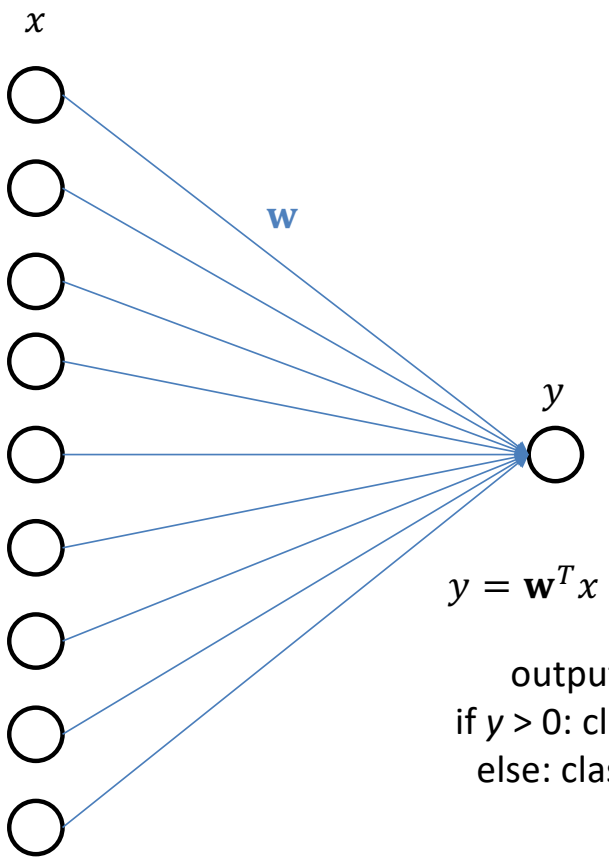


Neural Network Timeline



Source: Pumalin

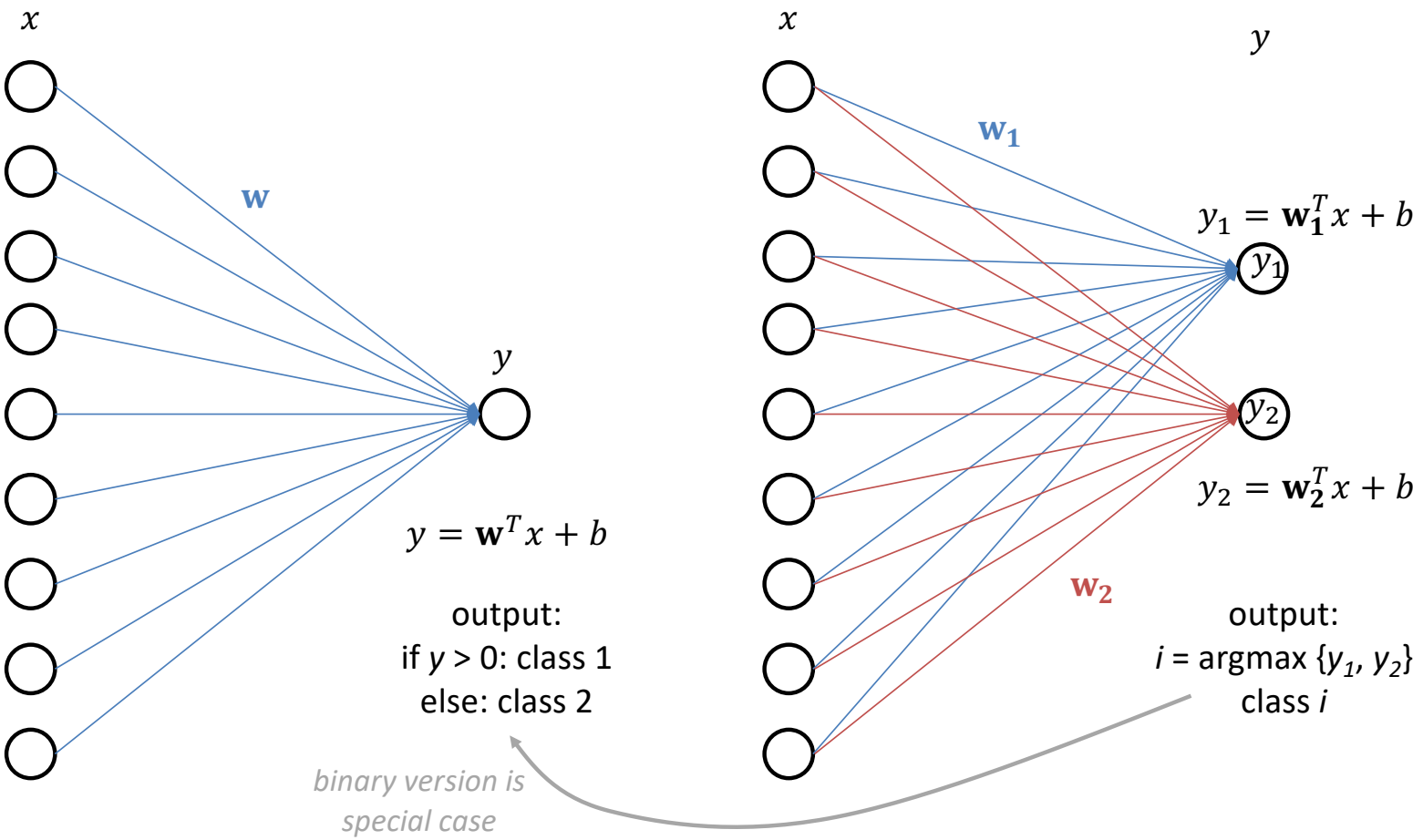
Remember Multi-class Linear Regression/Perceptron?



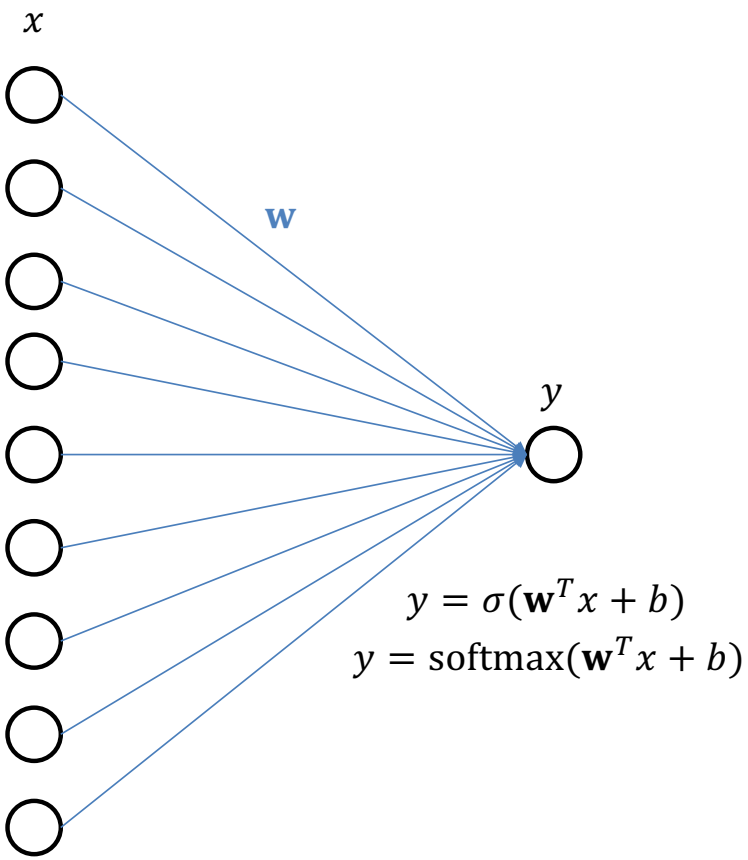
$y = \mathbf{w}^T x + b$

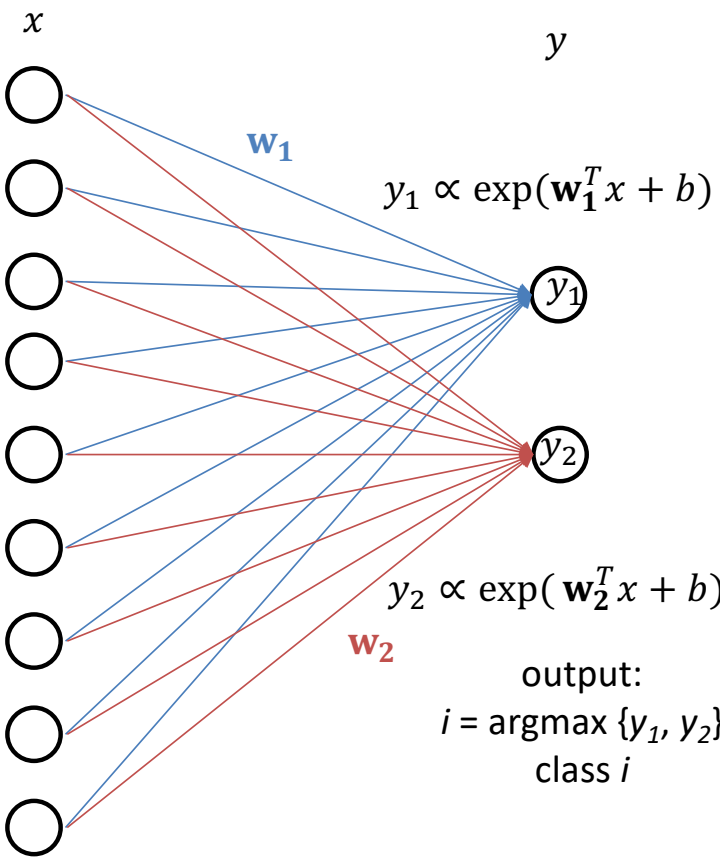
output:
if $y > 0$: class 1
else: class 2

Linear Regression/Perceptron: A Per-Class View



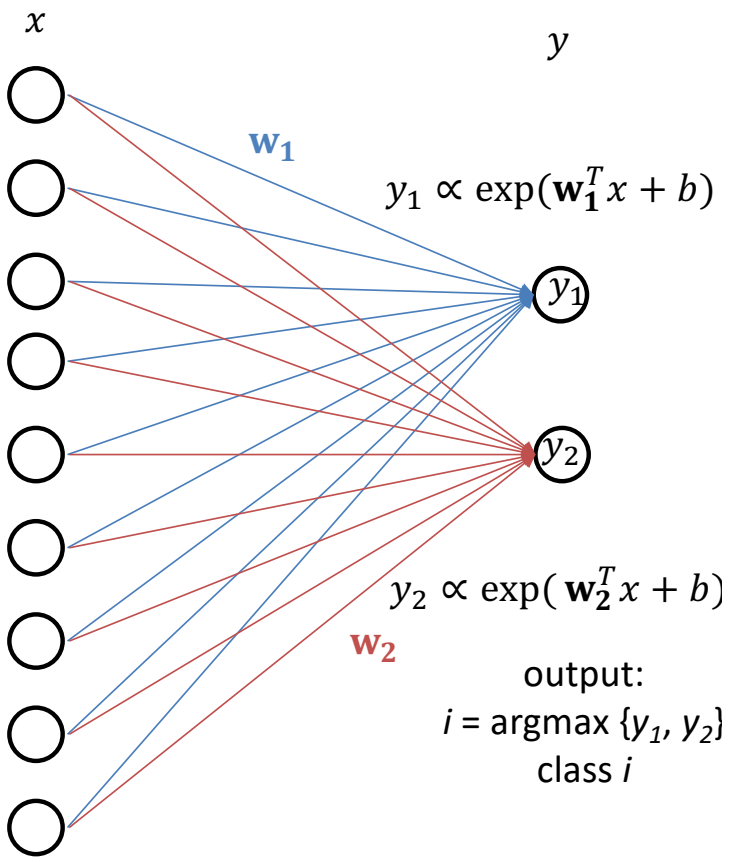
Logistic Regression/Classification





Logistic Regression/Classification

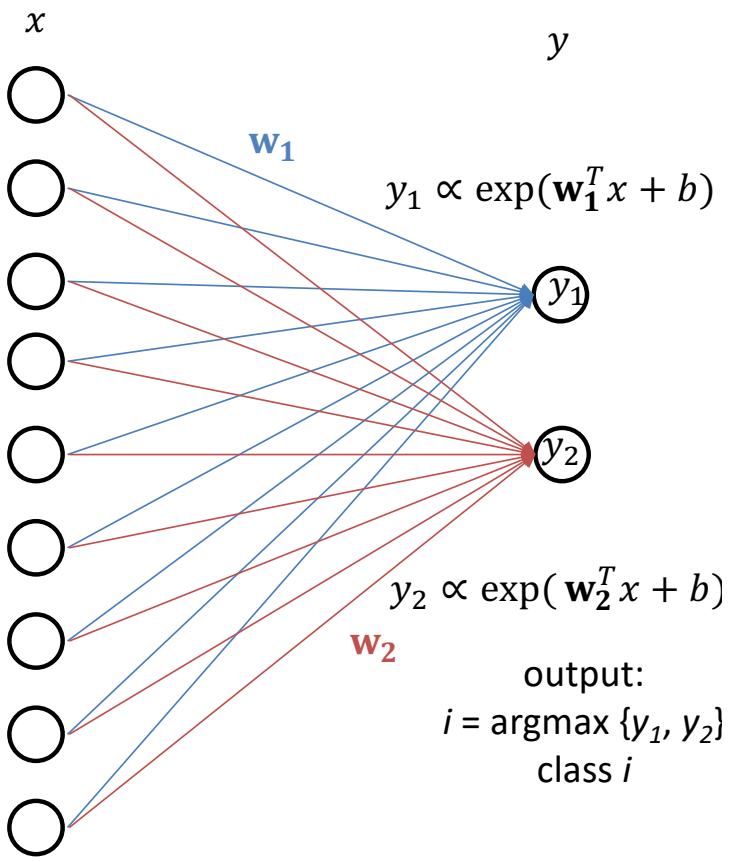
Q: Why didn't our maxent formulation from last class have multiple weight vectors?



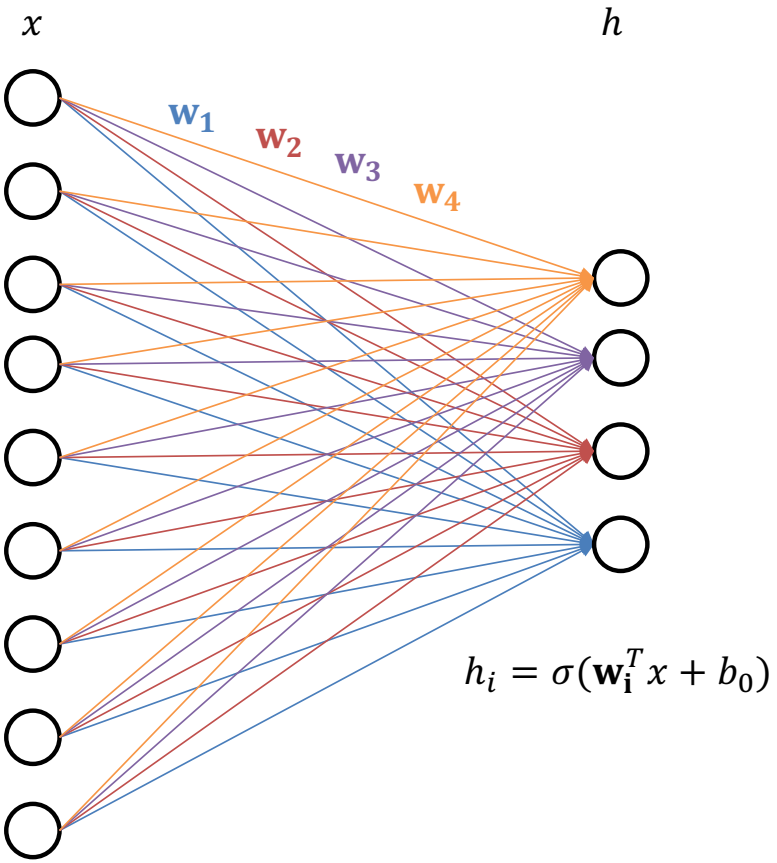
Logistic Regression/Classification

Q: Why didn't our maxent formulation from last class have multiple weight vectors?

A: Implicitly it did. Our formulation was $y \propto \exp(w^T f(x, y))$



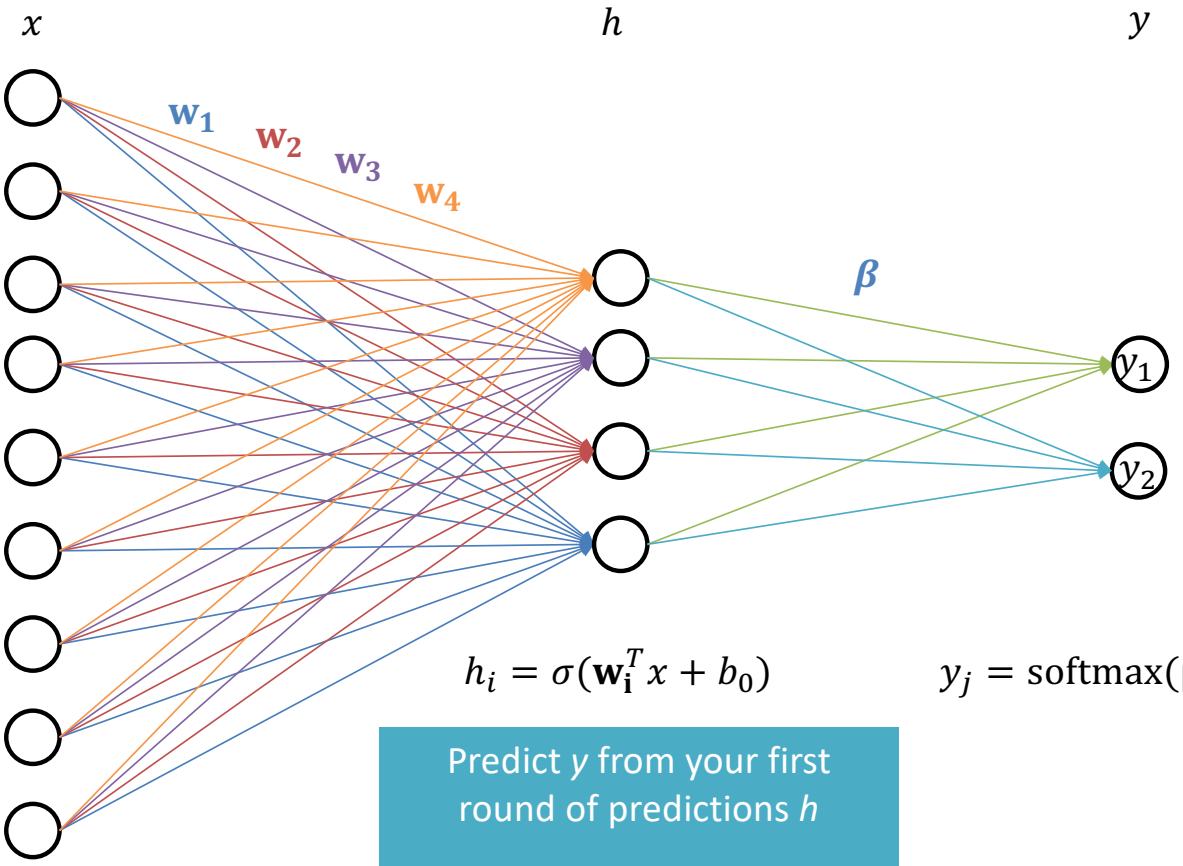
Stacking Logistic Regression



Goal: you still want to predict y

Idea: Can making an initial round of separate (independent) *binary* predictions *h* help?

Stacking Logistic Regression



$h_i = \sigma(\mathbf{w_i}^T x + b_0)$

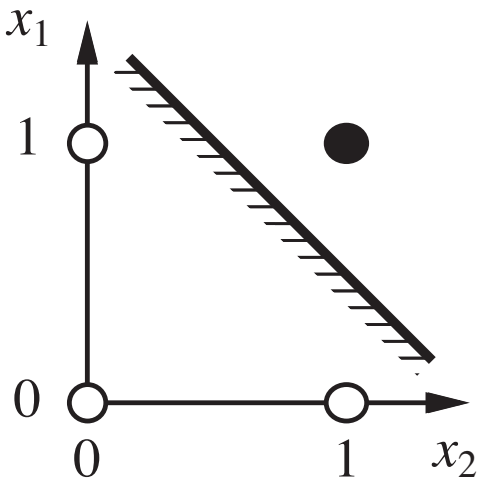
$y_j = \text{softmax}(\boldsymbol{\beta_j}^T h + b_1)$

Predict *y* from your first round of predictions *h*

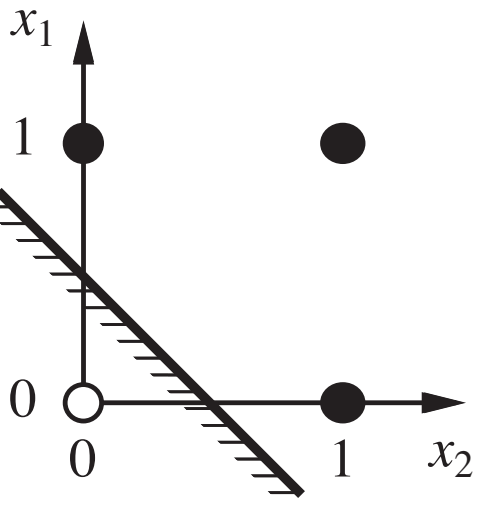
Idea: data/signal compression

Not with a perceptron ☹

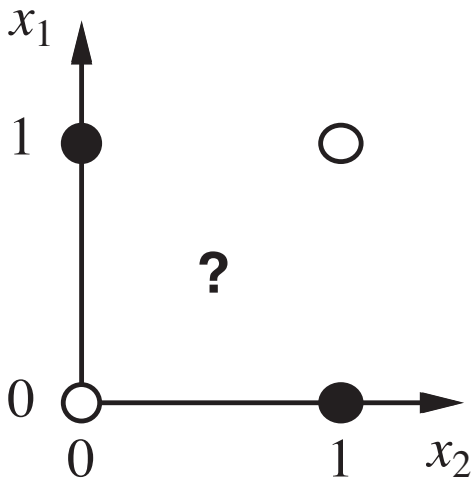
Consider Boolean operators (and, or, xor) with four possible inputs: 00 01 10 11



(a) x_1 **and** x_2



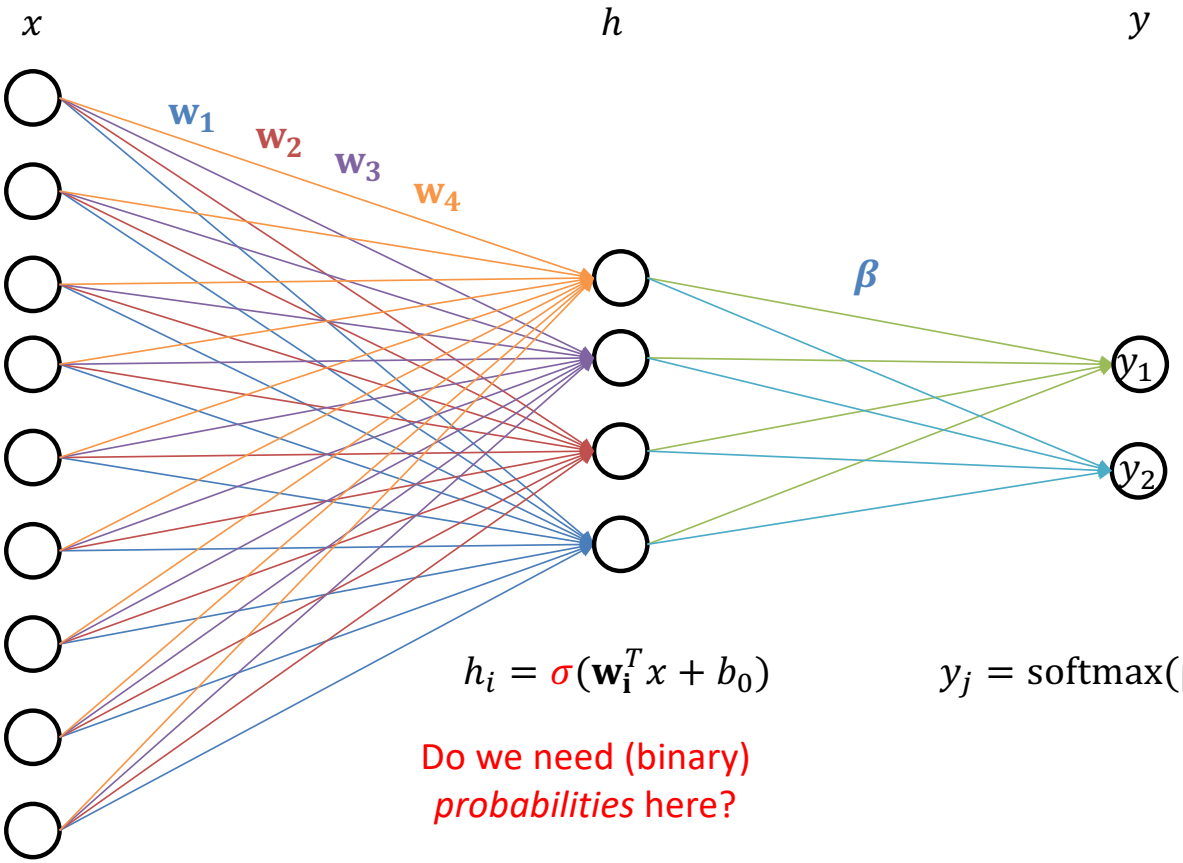
(b) x_1 **or** x_2



(c) x_1 **xor** x_2

Training examples are **not linearly separable** for one case: *sum=1 iff x1 xor x2*

Stacking Logistic Regression

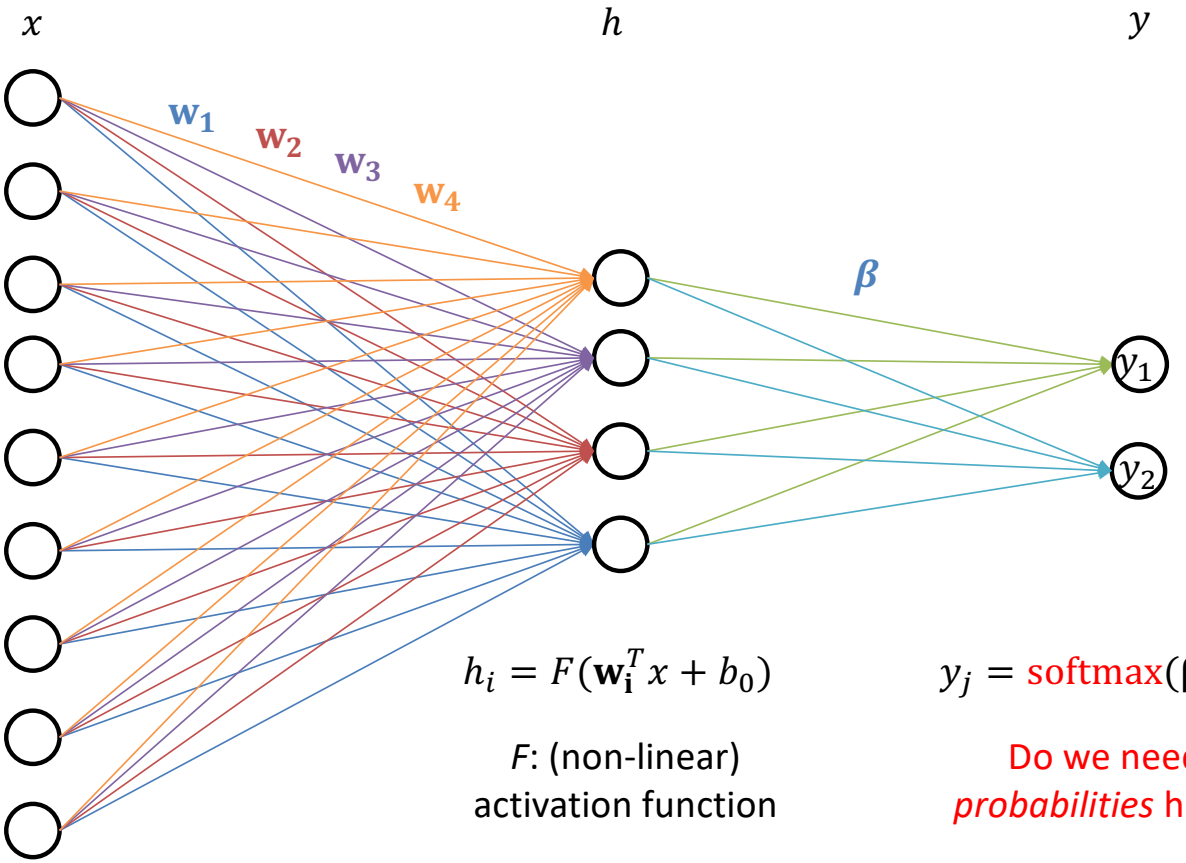


$h_i = \sigma(\mathbf{w_i}^T x + b_0)$

$y_j = \text{softmax}(\boldsymbol{\beta_j}^T h + b_1)$

Do we need (binary) *probabilities* here?

Stacking Logistic Regression



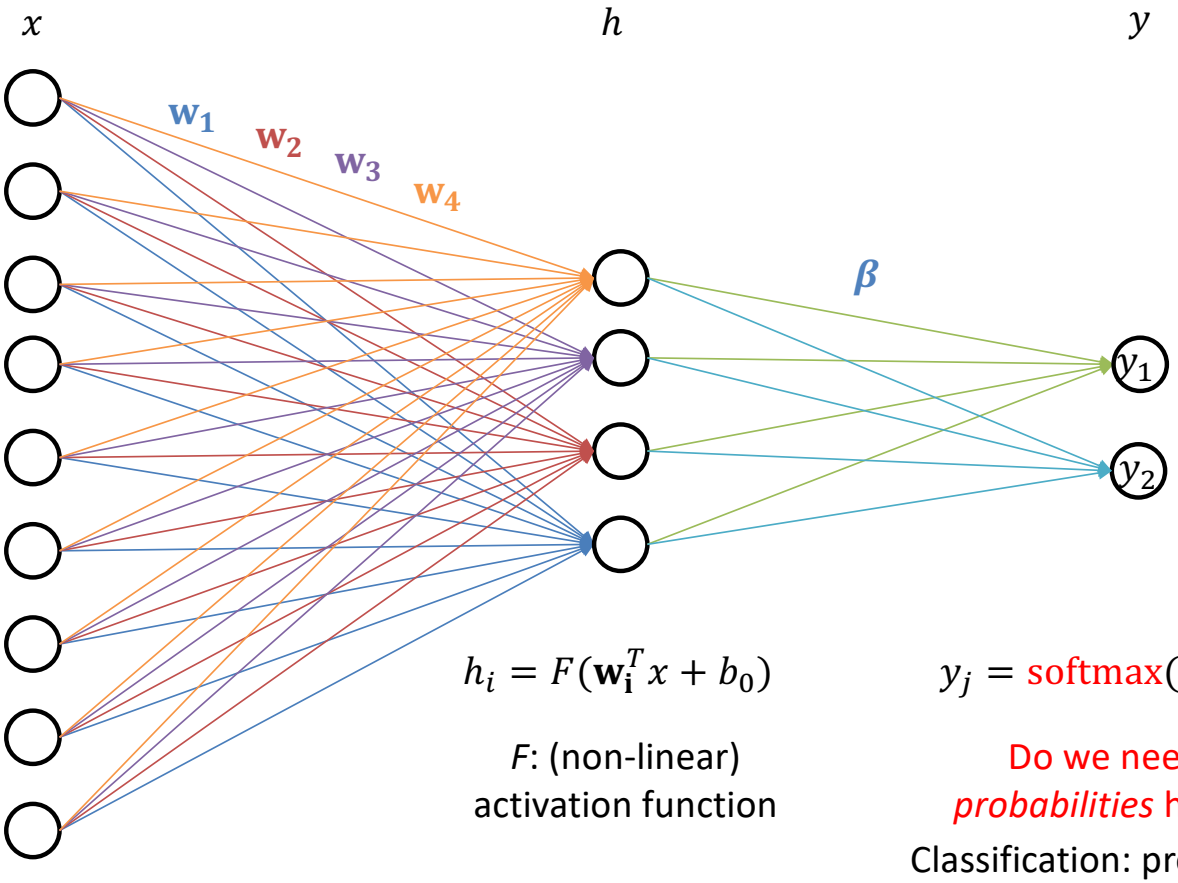
$$h_i = F(\mathbf{w_i}^T x + b_0)$$

F: (non-linear) activation function

$$y_j = \text{softmax}(\boldsymbol{\beta_j}^T h + b_1)$$

Do we need *probabilities* here?

Stacking Logistic Regression



$$h_i = F(\mathbf{w_i}^T x + b_0)$$

F: (non-linear) activation function

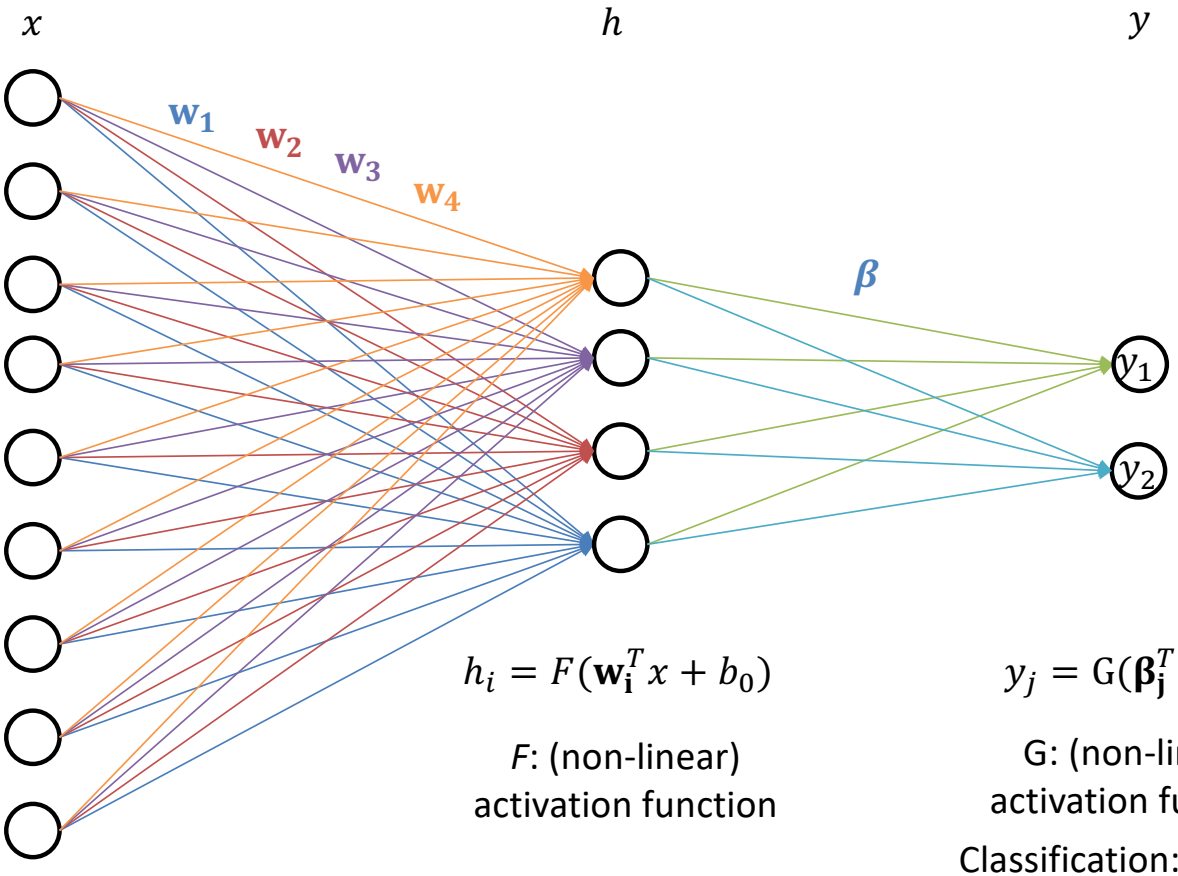
$$y_j = \text{softmax}(\boldsymbol{\beta_j}^T h + b_1)$$

Do we need *probabilities* here?

Classification: probably

Regression: not really

Stacking Logistic Regression



$h_i = F(\mathbf{w_i}^T x + b_0)$

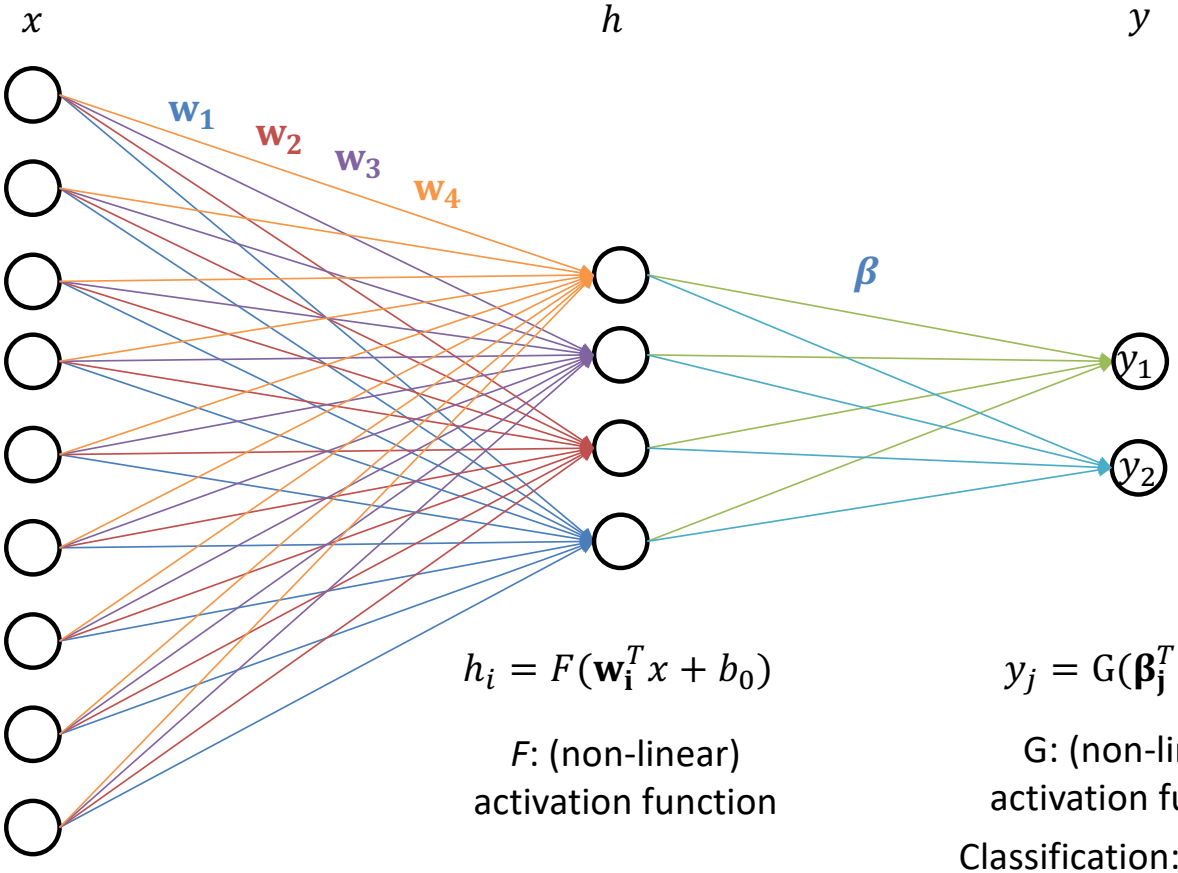
F: (non-linear) activation function

$y_j = \mathrm{G}(\boldsymbol{\beta_j}^T h + b_1)$

G: (non-linear) activation function

Classification: softmax
Regression: identity

Multilayer Perceptron, a.k.a. Feed-Forward Neural Network



$$h_i = F(\mathbf{w_i}^T x + b_0)$$

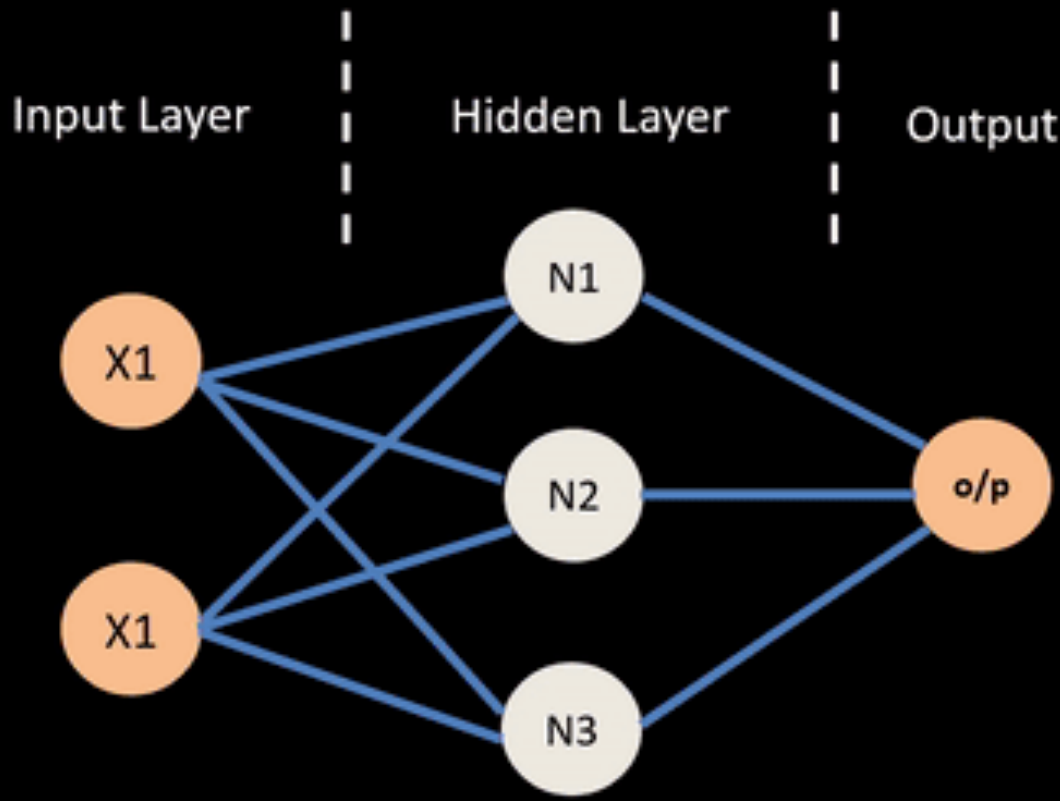
F: (non-linear) activation function

$$y_j = \mathrm{G}(\boldsymbol{\beta_j}^T h + b_1)$$

G: (non-linear) activation function

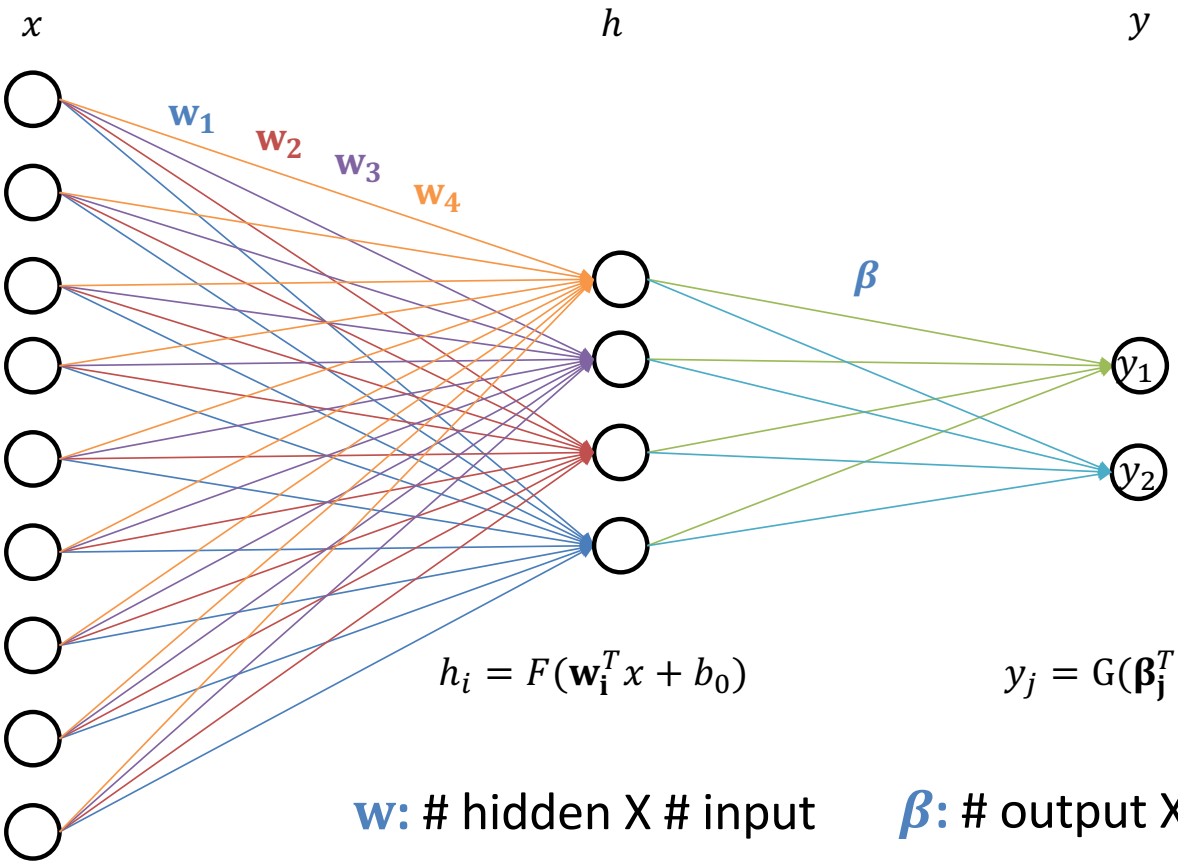
Classification: softmax
Regression: identity

Feed Forward Neural Network

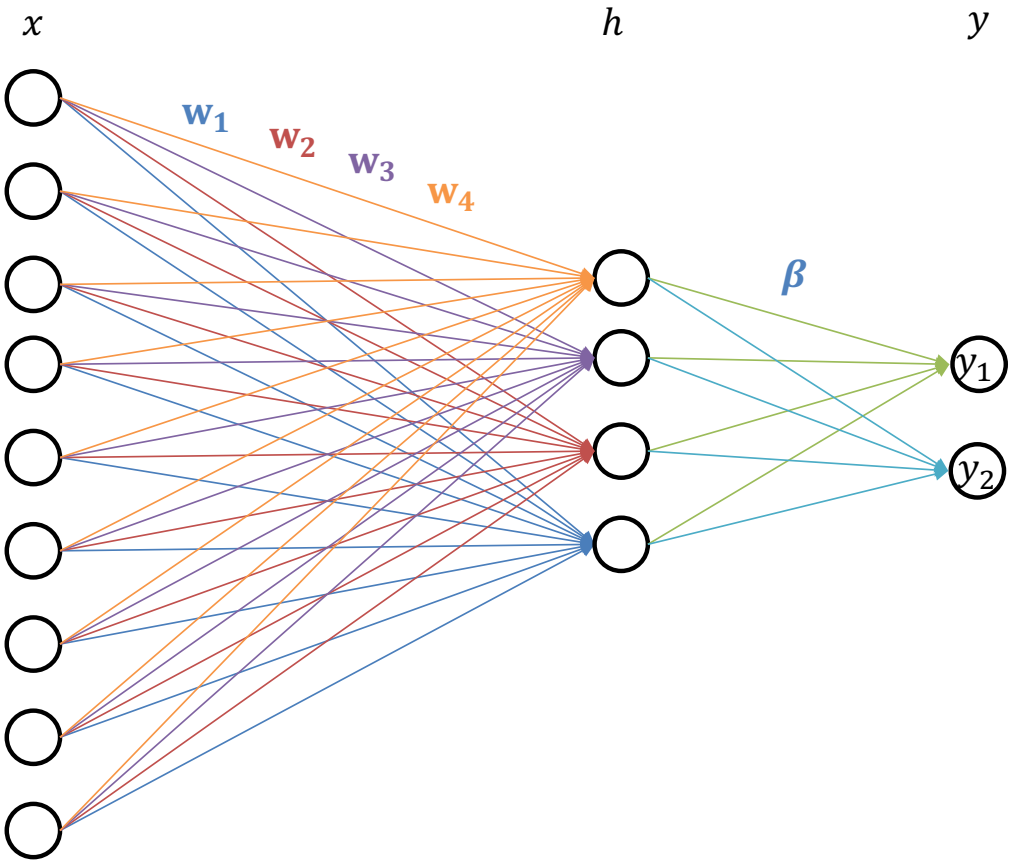


Information flows in forward direction only

Feed-Forward Neural Network



Why Non-Linear?

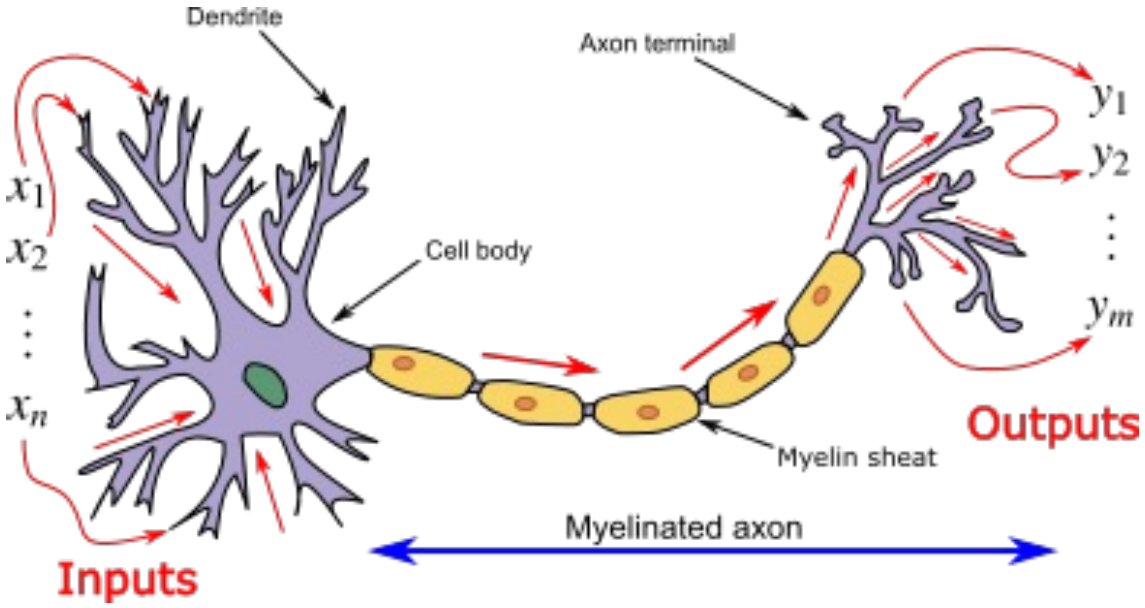


$$y_j = \mathrm{G}\left(\boldsymbol{\beta}_j^T h + b_1\right)$$

$$\Downarrow$$

$$y_j = G\left(\beta_j^T\left(F\left(w_i^T x + b_0\right)\right)_i\right)$$

Why called ‘Neural’ Network?



Neuron and myelinated axon, with signal flow from inputs at dendrites to outputs at axon terminals

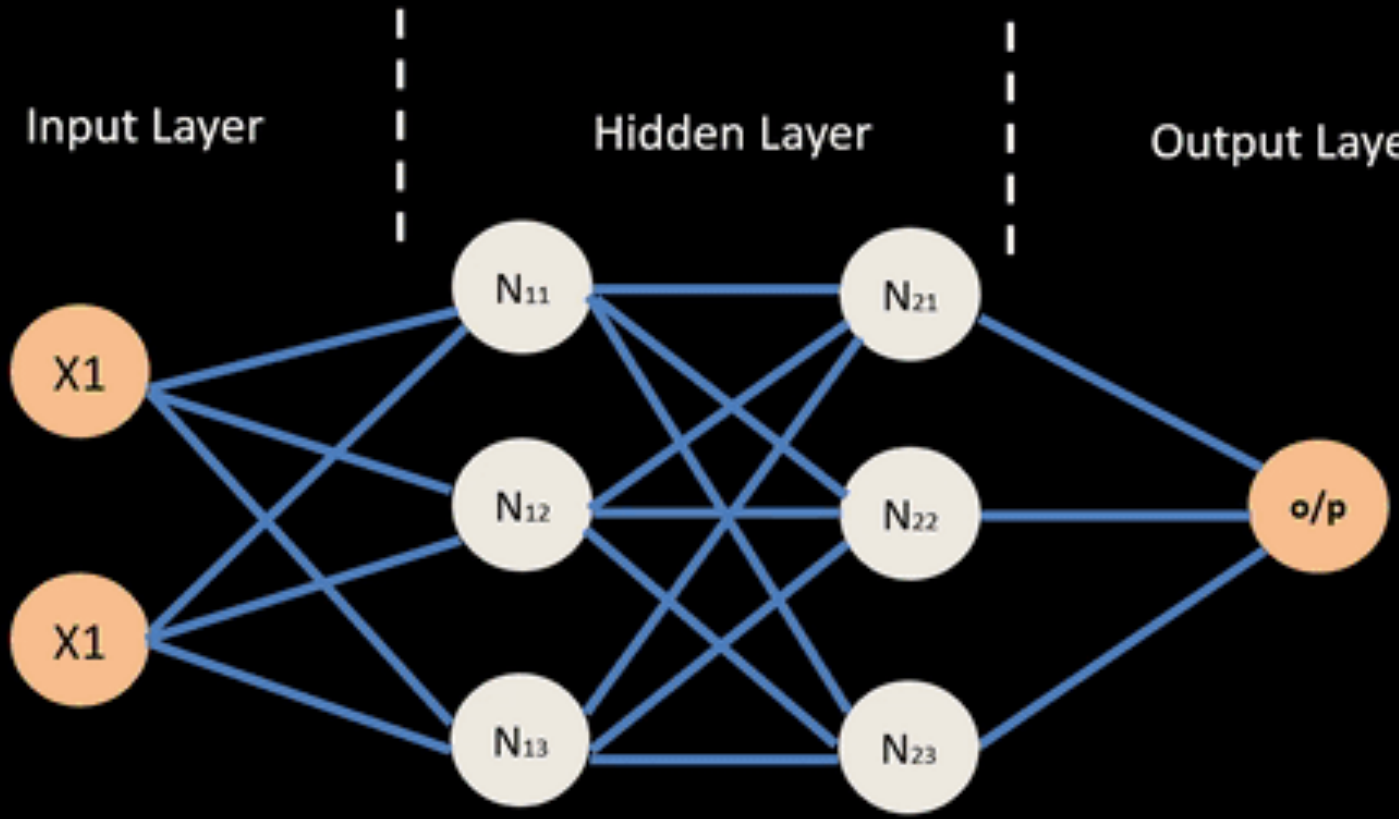
Neurons have body, axon and many dendrites

- In one of two states: firing and rest
- They fire if total incoming stimulus > threshold

Synapse: thin gap between axon of one neuron and dendrite of another

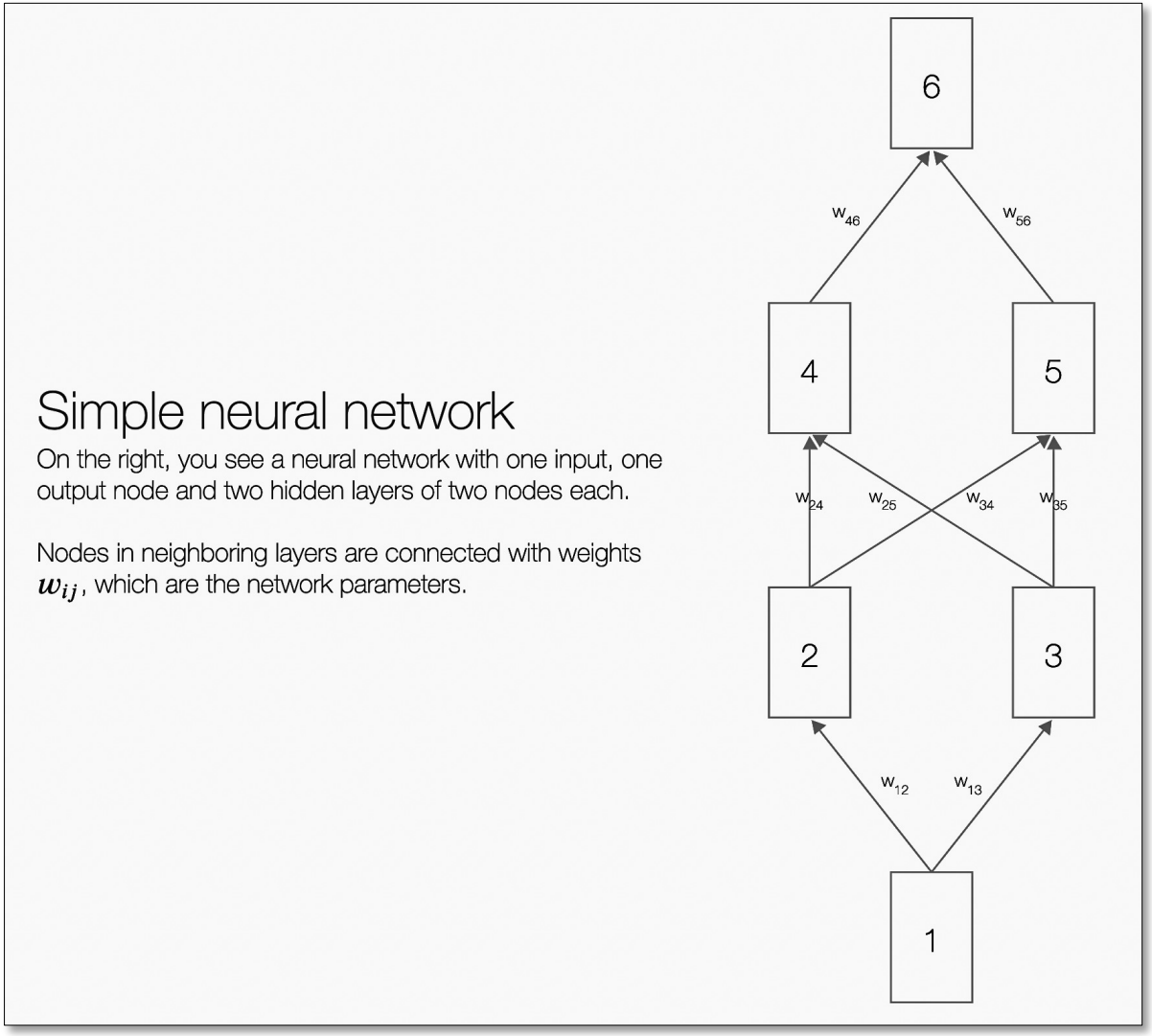
- Signal exchange

Neural Network – Backpropagation



Backpropagation Explained

Click on image (or here) for a simple interactive demo in your browser of how backpropagation updates weights in a neural network to reduce errors when processing training data



Universal Function Approximator

Theorem [Kurt Hornik et al., 1989]: Let F be a continuous function on a bounded subset of D-dimensional space. Then there exists a two-layer network G with finite number of hidden units that approximates F arbitrarily well. For all x in the domain of F, $|F(x) - G(x)| < \varepsilon$

"a two-layer network can approximate any function"

Going from one to two layers dramatically improves the representation power of the network

Slide courtesy Hamed Pirsiavash

How Deep Can They Be?

So many choices:

Architecture

# of hidden layers

# of units per hidden layer

Computational Issues:

Vanishing gradients

Gradients shrink as one moves away from the output layer

Convergence is slow

Opportunities:

Training deep networks is an active area of research

Layer-wise initialization (perhaps using unsupervised data)

Engineering: GPUs to train on massive labelled datasets

Slide courtesy Hamed Pirsiavash

Some Results: Digit Classification

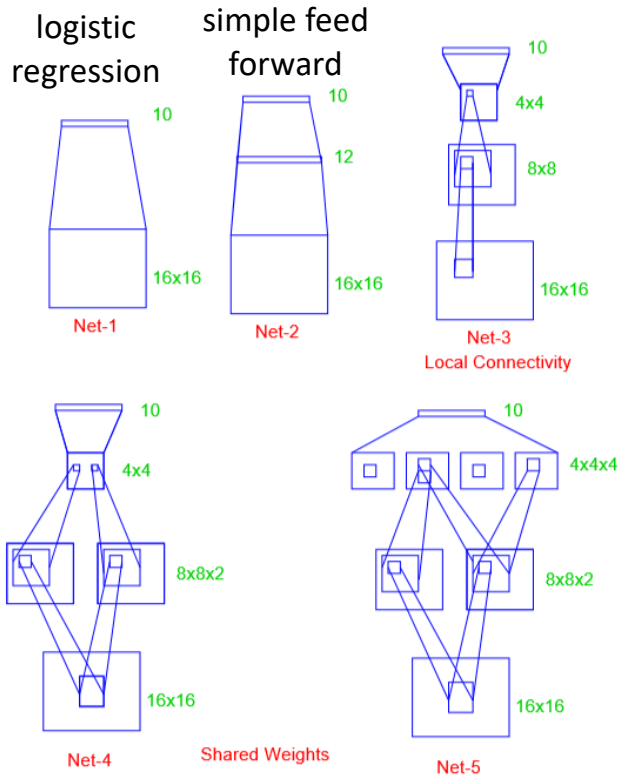


FIGURE 11.10. *Architecture of the five networks used in the ZIP code example.*

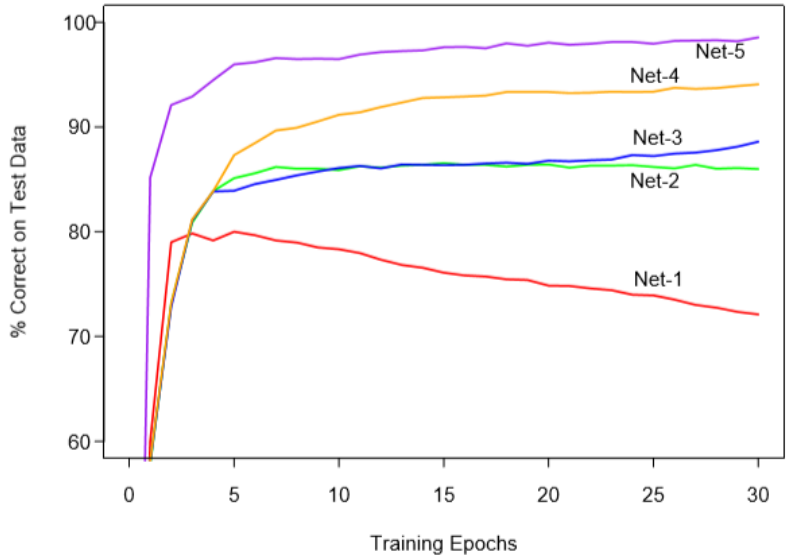
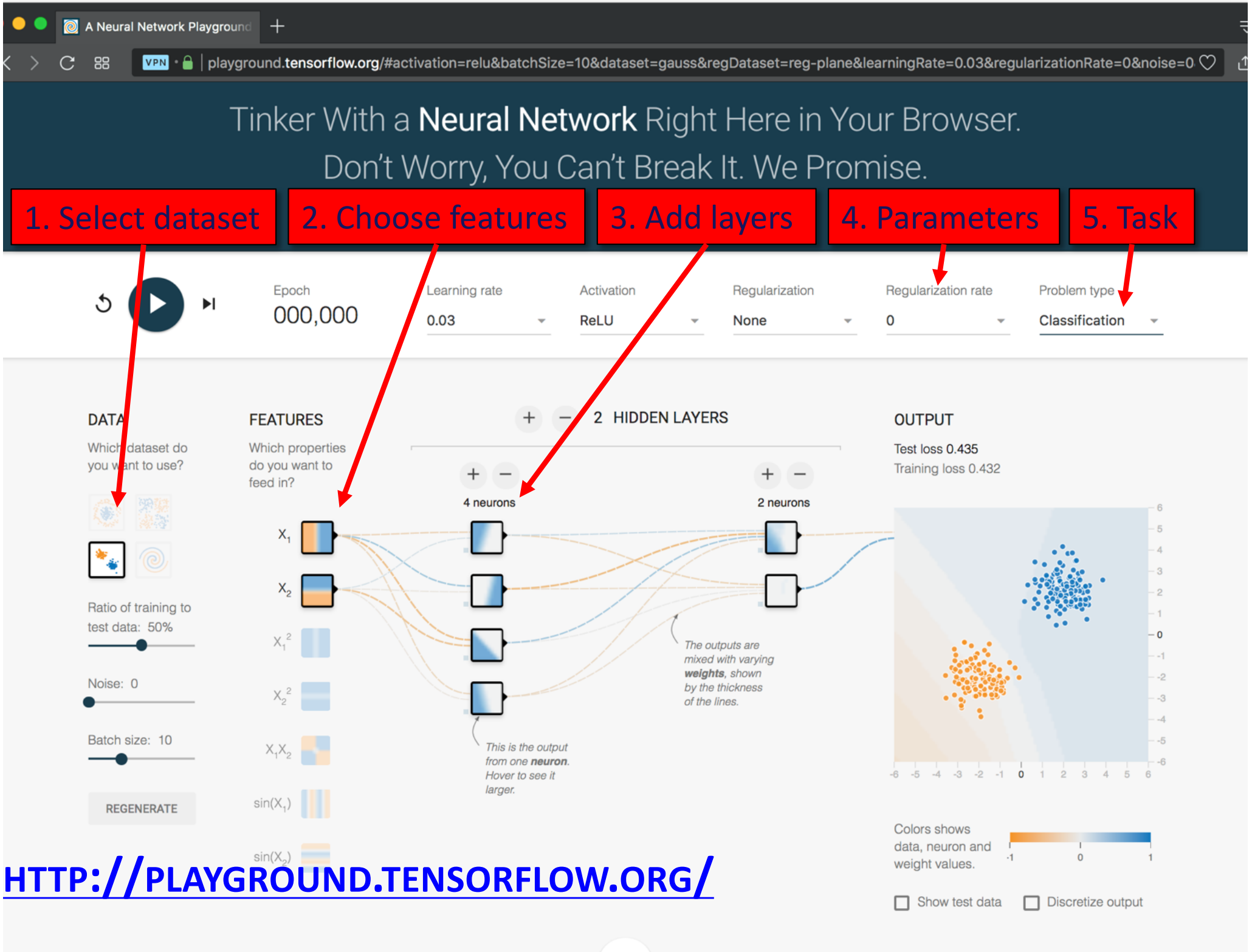


FIGURE 11.11. *Test performance curves, as a function of the number of training epochs, for the five networks of Table 11.1 applied to the ZIP code data.*

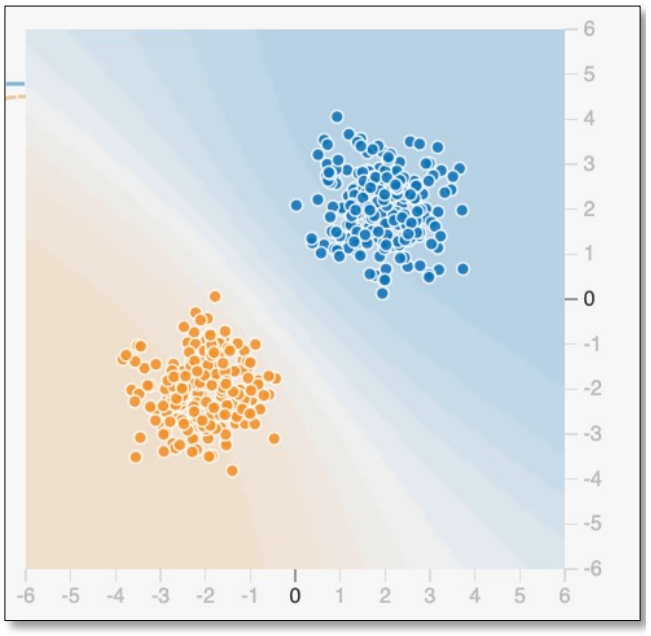
(similar to MNIST in A2, but not exactly the same)


A Neural Network Playground
playground.tensorflow.org/#activation=relu&batchSize=10&dataset=gauss&regDataset=reg-plane&learningRate=0.03&regularizationRate=0&noise=0
Tinker With a Neural Network Right Here in Your Browser.
Don't Worry, You Can't Break It. We Promise.
1. Select dataset
2. Choose features
3. Add layers
4. Parameters
5. Task
Epoch
000,000
Learning rate
0.03
Activation
ReLU
Regularization
None
Regularization rate
0
Problem type
Classification
DATA
Which dataset do you want to use?
Ratio of training to test data: 50%
Noise: 0
Batch size: 10
REGENERATE
FEATURES
Which properties do you want to feed in?
2 HIDDEN LAYERS
4 neurons
2 neurons
This is the output from one neuron. Hover to see it larger.
The outputs are mixed with varying weights, shown by the thickness of the lines.
OUTPUT
Test loss 0.435
Training loss 0.432
Colors shows data, neuron and weight values.
Show test data
Discretize output
HTTP://PLAYGROUND.TENSORFLOW.ORG/

TensorFlow Playground

- Great javascript app demonstrating many basic neural network concepts (e.g., MLPs)
- Doesn't use TensorFlow software, but a lightweight js library
- Runs in a Web browser
- See **http://playground.tensorflow.org/**
- Code also available on GitHub
- Try the playground exercises in Google's machine learning crash course

Datasets



- Six datasets, each with 500 (x,y) points on a plane where x and y between -5 and +5
- Points have *labels* of positive (orange) or negative (blue)
- Two possible machine learning *tasks*:
  - Classification: Predict class of test points
  - Regression: find function to separate classes
- *Evaluation*: split dataset into training and test, e.g., 80% training, 20% test

Available Input features

X_1 Point's x value

X_2 Point's y value

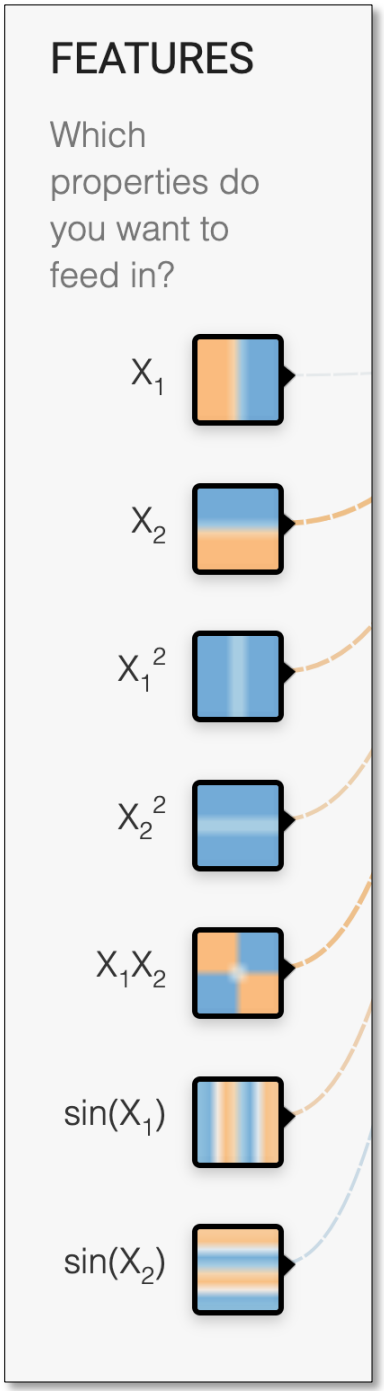
X_1^2 Point's x value squared

X_2^2 Point's y value squared

X_1X_2 Product of point's x & y values

$\sin(X_1)$ Sine of point's x value

$\sin(X_2)$ Sine of point's y value



Designing a neural network

- Simple feed forward NNs have a few choices
  - What input **features** to use
  - How many **hidden layers** and their details**:**
    - How many neurons are in each layer
    - How each layer is connected to ones before & after
- Complex NNs have more choices
  - E.g., CNNs, RNNs, etc.
- High-level interfaces (e.g., Keras) try to make this easier

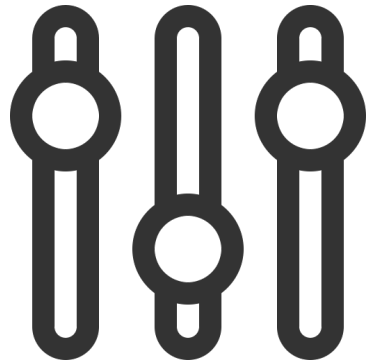
Training a Neural Network

- Neural networks are used for **supervised** machine learning and need to be trained
- The training process is broken done in a series of ***epochs***
  - In each epoch, all training data run through the system to adjust the NN parameters
- Process ends after a fixed # of epochs or when **error rate** flattens or starts increasing

Typical Training Flow

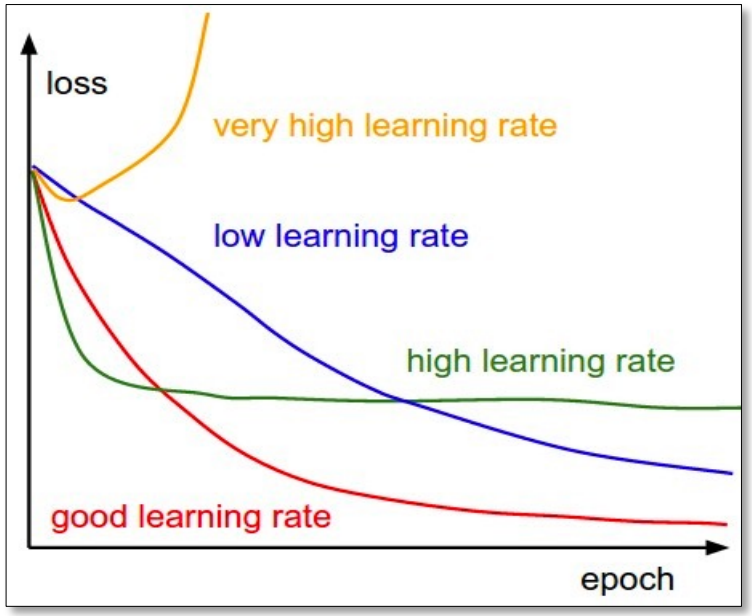
- Divide training data into batches of instances (e.g., batch size = 10)
- For each epoch:
  - For each batch:
    - Instances run through network, noting difference between predicted and actual value
    - Backpropagation used to adjust connection weights
  - Stop when training loss flatten out
- If test loss is too high, then try
  - Adding additional hidden layers
  - Adding more features to inputs
  - Adjusting hyperparameters (e.g., learning rate)
  - Get more training data

Hyperparameters



- Parameters whose values are set before the learning process begins
- Basic neural network hyperparameters
  - Learning rate (e.g., 0.03)
  - Activation function (e.g., ReLU)
  - Regularization (e.g., L2)
  - Regularization rate (e.g., 0.1)

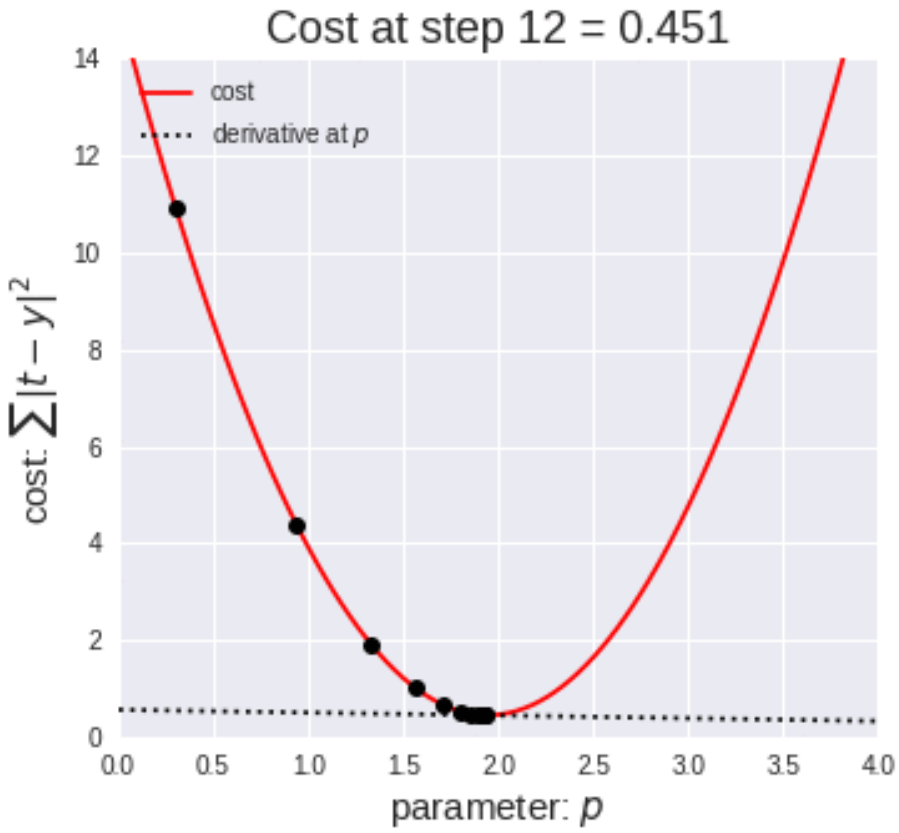
Learning rate



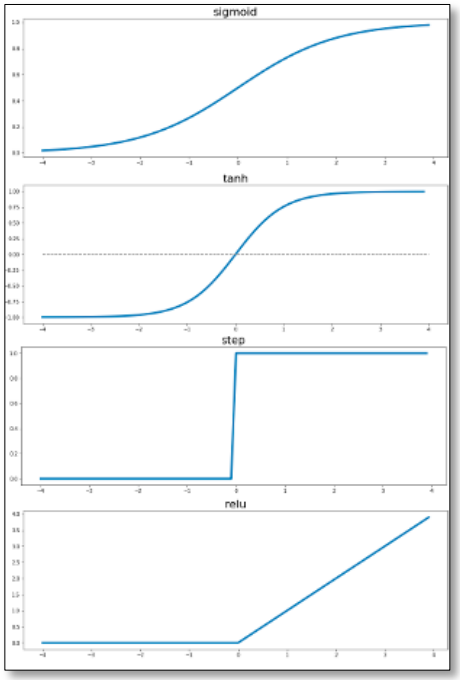
- Gradient descent used in backpropagation to adjust weights to minimize the loss function
- Learning rate determines how quickly weights are adjusted each time
- If too high, we may miss some or most minima
  - Result: erratic performance or never achieving a low loss
- If too low, learning will take longer than necessary

Gradient Descent

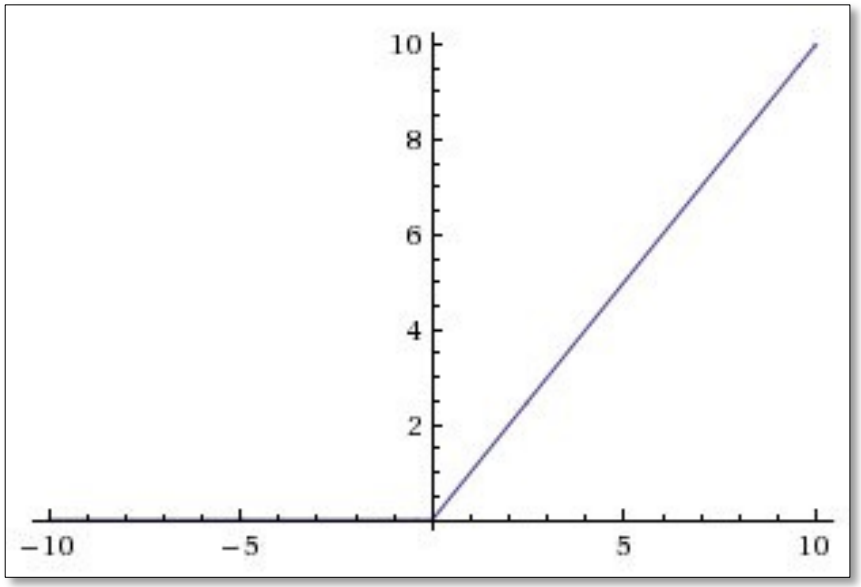
- Iterative process used in ML to find local minimum in our loss function measuring errors
- Moves in direction of steepest descent
- Step size decreases as steepness lessens to avoid missing minima
- Custom variants for NNs include adam optimization



Activation Function

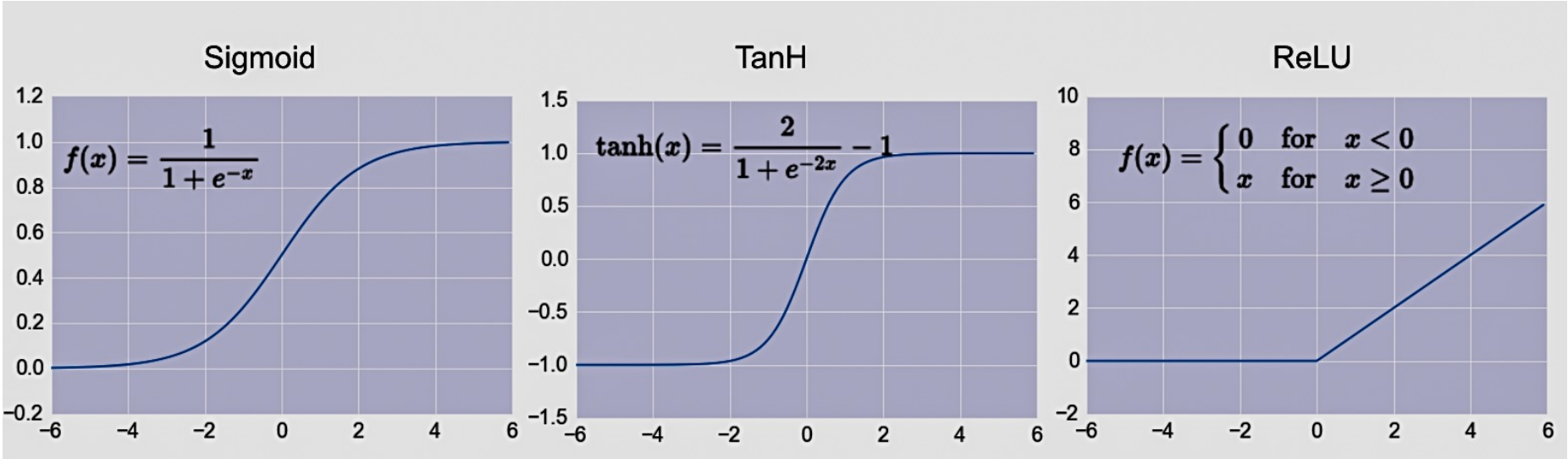


- Determines a node's output given weighted sum of its inputs
- The ReLu (rectified linear unit) is simple and a good choice for most networks
- Returns zero for negative values and its input for positive ones
  - f(x) = max(0,x)



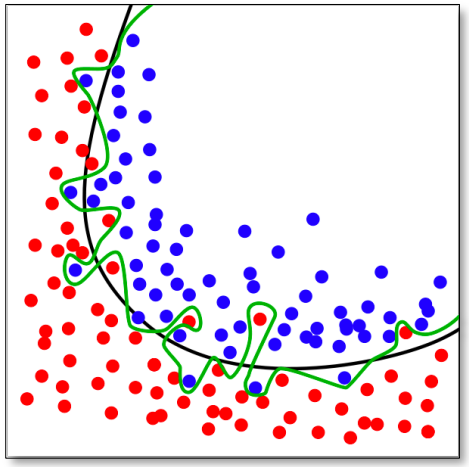
Common Activation Functions

- Define the output of a node given an input
- Very simple functions!



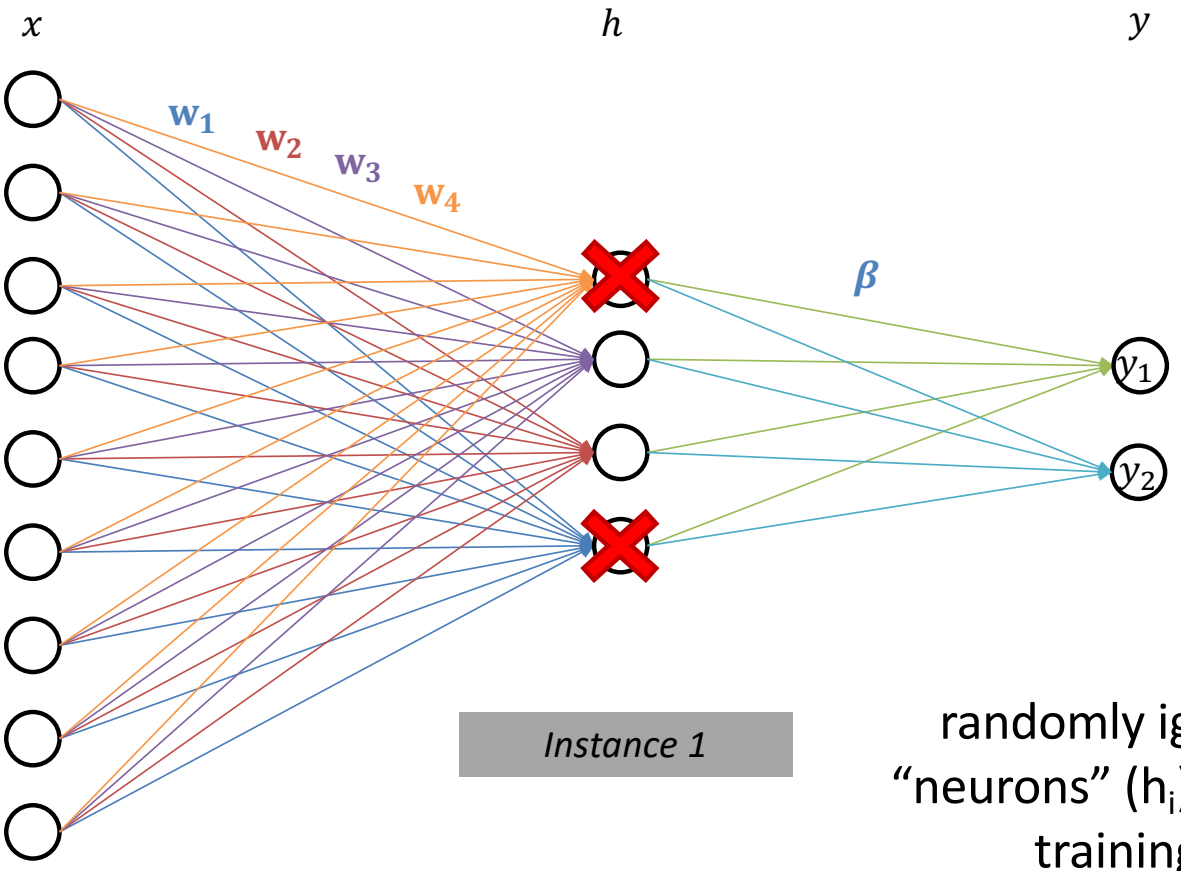
- Choice of activation function depends on problem and available computational power
- Comprehensive list of activation functions
- In practice

Regularization



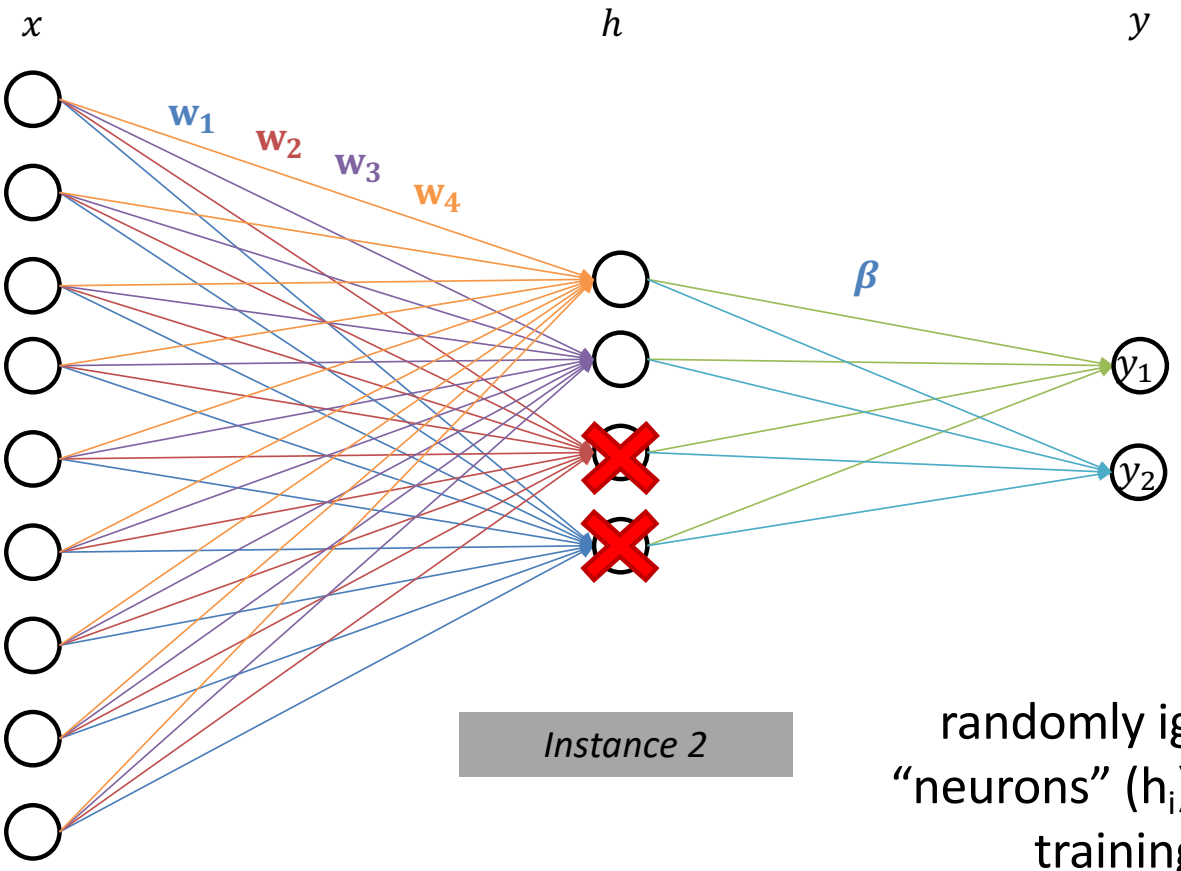
- Parameter to control overfitting,
  i.e. when the model does well on training data but poorly on new, unseen data
- L2 regularization is the most common
- Using dropout is another common way of reducing overfitting in neural networks
  - At each training stage, some nodes in hidden layer temporarily removed (dropped out)

Dropout: Regularization in Neural Networks



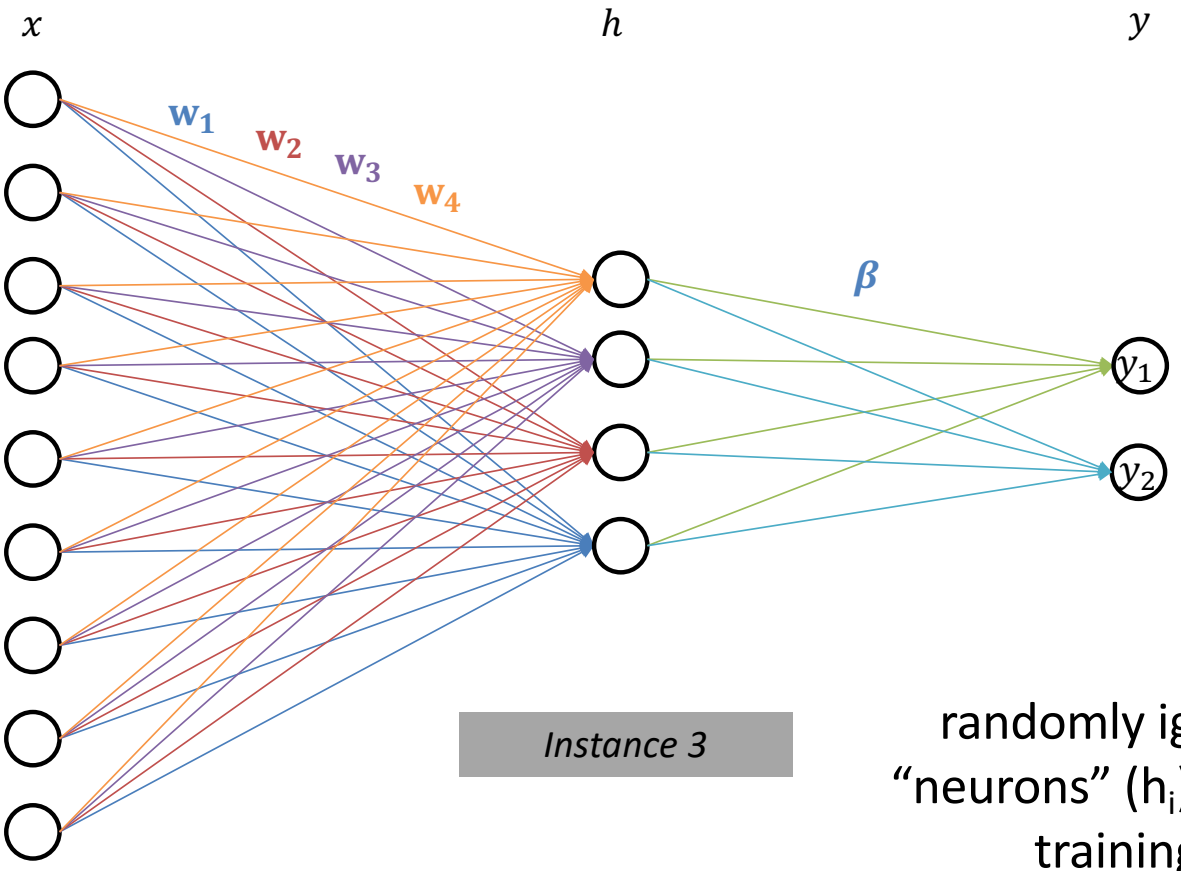
randomly ignore "neurons" (h_i) during training

Dropout: Regularization in Neural Networks



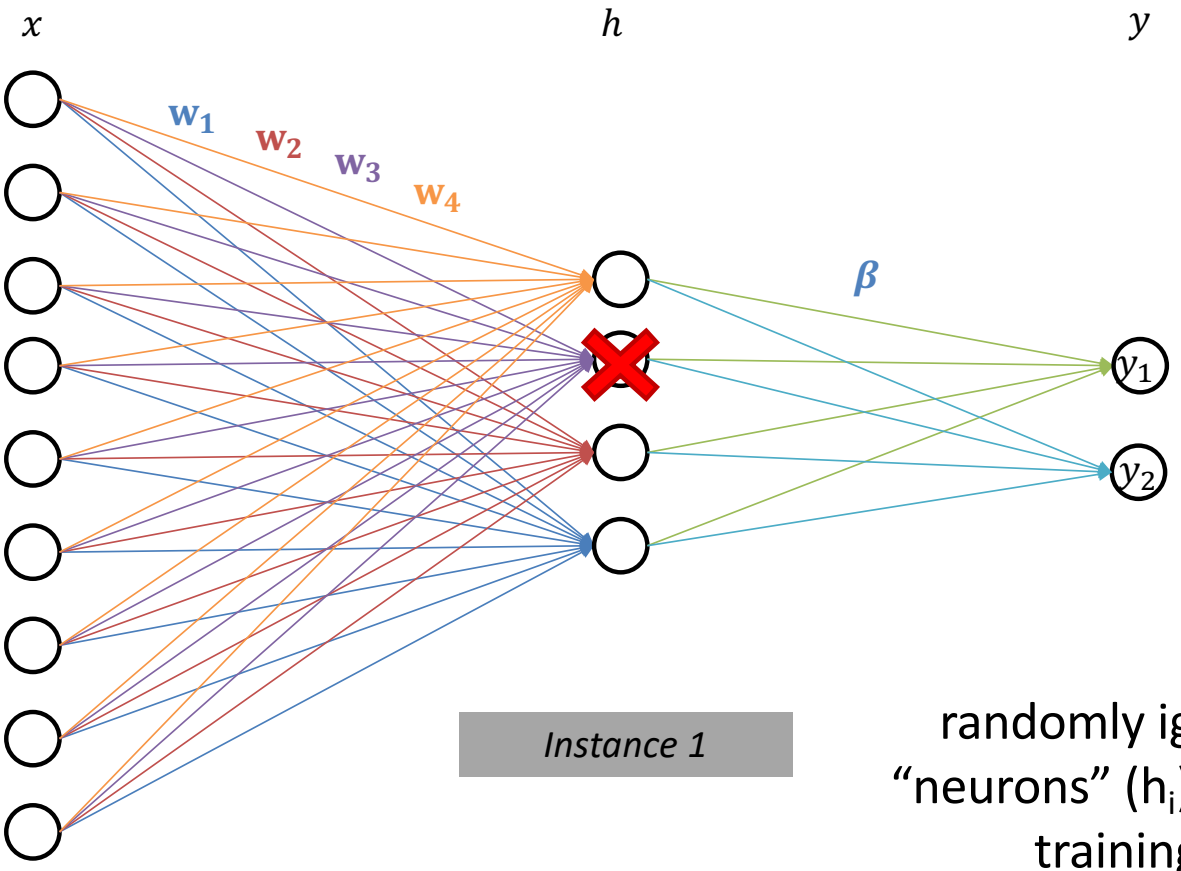
randomly ignore "neurons" (h_i) during training

Dropout: Regularization in Neural Networks



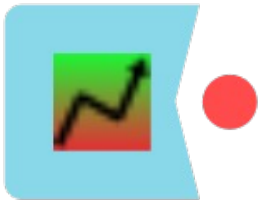
randomly ignore "neurons" (h_i) during training

Dropout: Regularization in Neural Networks

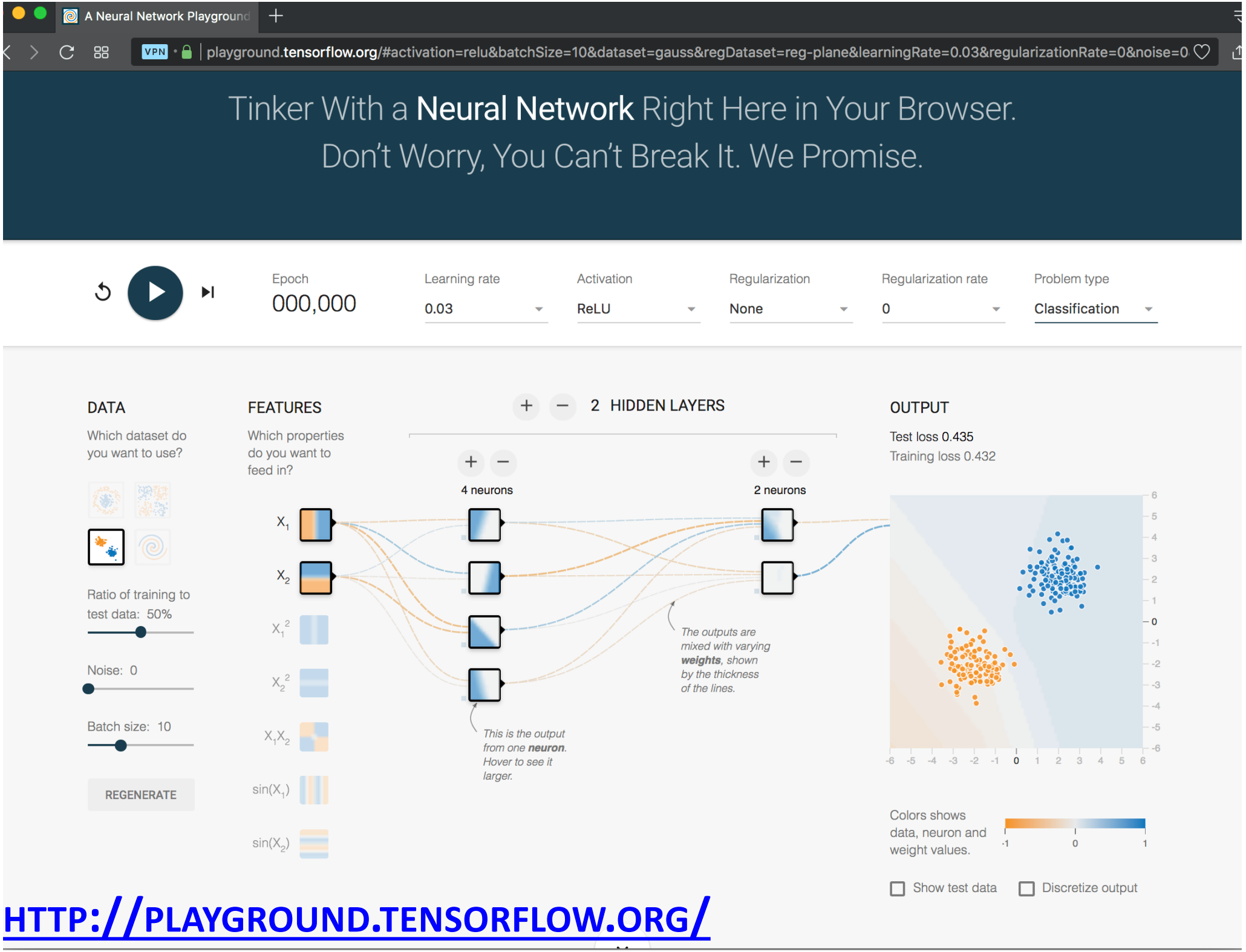


randomly ignore "neurons" (h_i) during training

Hyperparameter optimization



- How do we find the best settings for these hyperparameters?
- Experimentation
  - Experiment with a range of different settings (e.g., for learning rate) via multiple runs
  - Use a grid search tool, e.g., scikit learn's
- Experience
  - Similar problems with similar data will probably benefit from similar settings


A Neural Network Playground
playground.tensorflow.org/#activation=relu&batchSize=10&dataset=gauss&regDataset=reg-plane&learningRate=0.03&regularizationRate=0&noise=0
Tinker With a Neural Network Right Here in Your Browser.
Don't Worry, You Can't Break It. We Promise.
Epoch
000,000
Learning rate
0.03
Activation
ReLU
Regularization
None
Regularization rate
0
Problem type
Classification
DATA
Which dataset do you want to use?
Ratio of training to test data: 50%
Noise: 0
Batch size: 10
REGENERATE
FEATURES
Which properties do you want to feed in?
X1
X2
X1²
X2²
X1X2
sin(X1)
sin(X2)
2 HIDDEN LAYERS
4 neurons
2 neurons
The outputs are mixed with varying weights, shown by the thickness of the lines.
This is the output from one neuron. Hover to see it larger.
OUTPUT
Test loss 0.435
Training loss 0.432
Colors shows data, neuron and weight values.
Show test data
Discretize output
HTTP://PLAYGROUND.TENSORFLOW.ORG/

Deep Learning Frameworks (1)

- Popular open-source deep learning frame-works use Python at top-level; C++ in backend
  - TensorFlow (via Google)
  - Keras (Open Source, now TensorFlow's I/F)
  - PyTorch (via Facebook)
  - MxNet (Apache)
  - Caffe (Berkeley)
- TensorFlow and PyTorch now dominate; both make it easy to specify a complicated network
- Keras is now part of TensorFLow

Keras

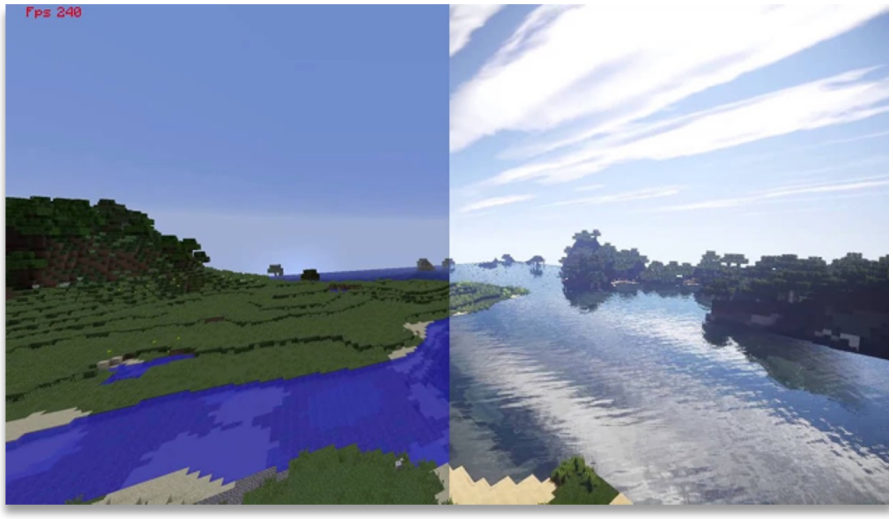
- "Deep learning for humans"
- A popular API works with TensorFlow provides good support at architecture level
- Keras (v2.4 +) only supports TensorFlow
- Supports CNNs and RNNs and common utility layers like dropout, batch normalization and pooling
- Coding neural networks used to be harder; Keras made it easier and more accessible
- Documentation: https://keras.io/

mo

But problems remained …

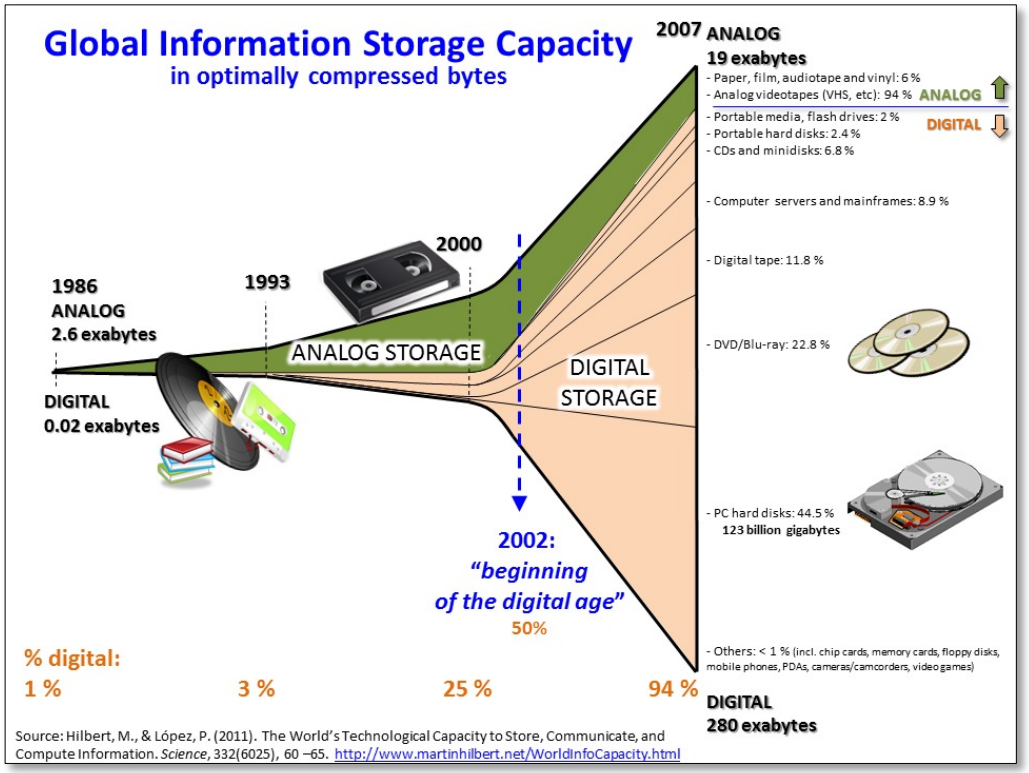
- It's often the case that solving a problem just reveals a new one that needs solving
- For a large MLPs, backpropagation takes forever to converge!
- Two issues:
  - Not enough compute power to train the model
  - Not enough labeled data to train the neural net
- SVMs may be better, since they converge to global optimum in O(n^2)

GPUs solve compute power problem



- GPUs (Graphical Processing Units) became popular in the 1990s to handle computing needed for better computer graphics
- GPUs are SIMD (single instruction, multiple data) processors
- Cheap, fast, and easy to program
- GPUs can do matrix multiplication and other matrix computations very fast

Need lots of data!



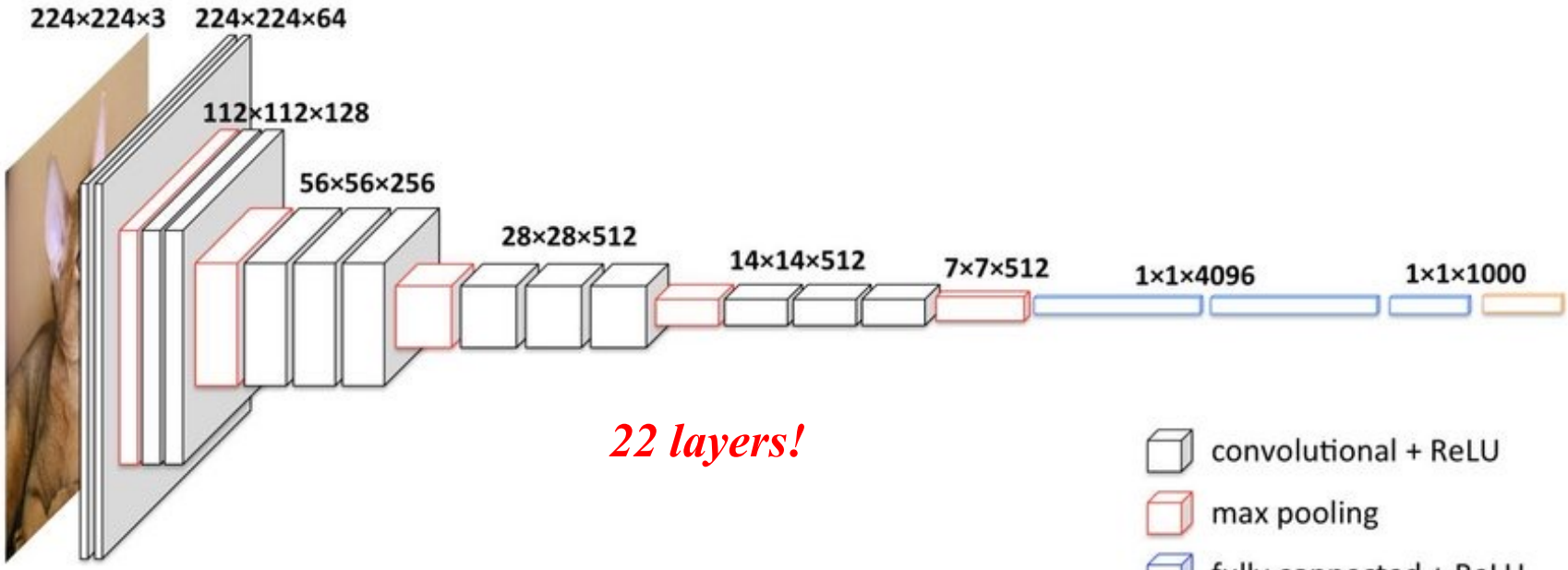
- 2000s introduced [big data]
- Cheaper storage
- Parallel processing (e.g., MapReduce, Hadoop, Spark, grid computing)
- Data sharing via the Web
  - Lots of images, many with captions
  - Lots of text, some with labels
- Crowdsourcing systems (e.g., [Mechanical Turk]) provided a way to get more human annotations

New problems are surfaced

- 2010s was a decade of domain applications
- These came with new problems, e.g.,
  - Images are too highly dimensioned!
  - Variable-length problems cause gradient problems
  - Training data is rarely labeled
  - Neural nets are uninterpretable
  - Training complex models required days or weeks
- This led to many new “deep learning” neural network models

Deep Learning

- Deep learning refers to models going beyond simple feed-forward multi-level perceptron
  - Though it was used in a ML context as early as 1986
- "deep" refers to the models having many layers (e.g., 10-20) that do different things



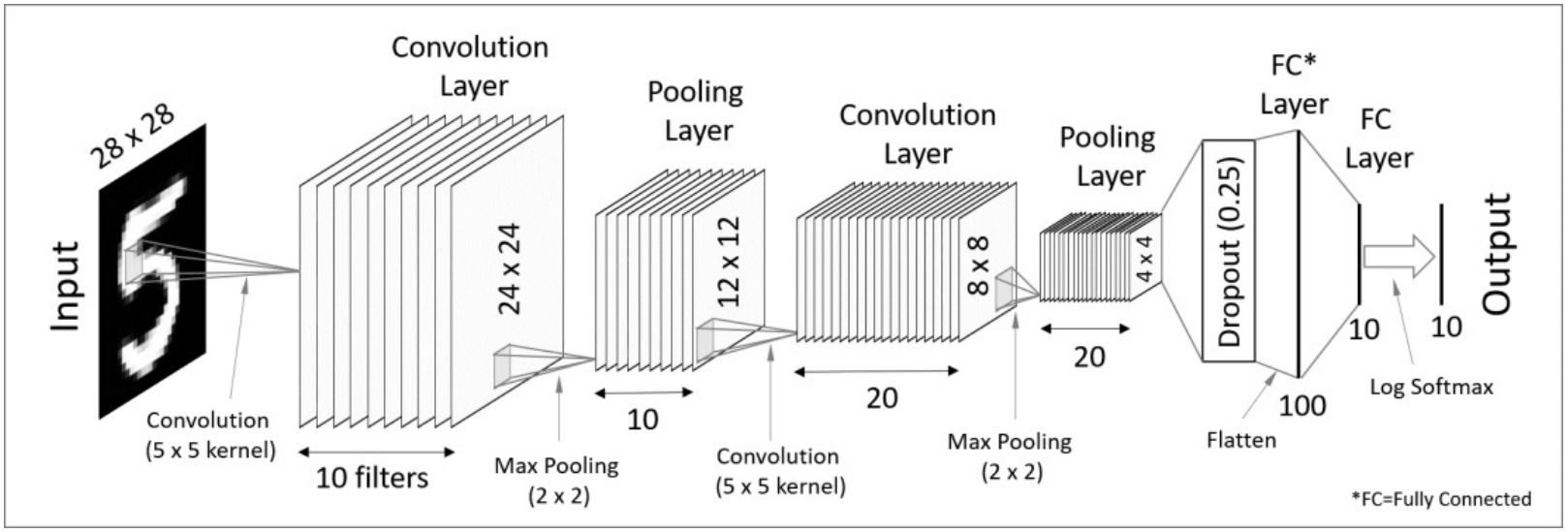
The VGG16 CNN model for image processing

Neural Network Architectures

Current focus on large networks with different "architectures" suited for different tasks

- Feedforward Neural Network
- CNN: **C**onvolutional **N**eural **N**etwork
- RNN: **R**ecurrent **N**eural **N**etwork
- LSTM: **L**ong Short **T**erm **M**emory
- GAN: **G**enerative **A**dversarial **N**etwork
- Transformers: generating output sequence from input sequence

CNN: Convolutional Neural Network



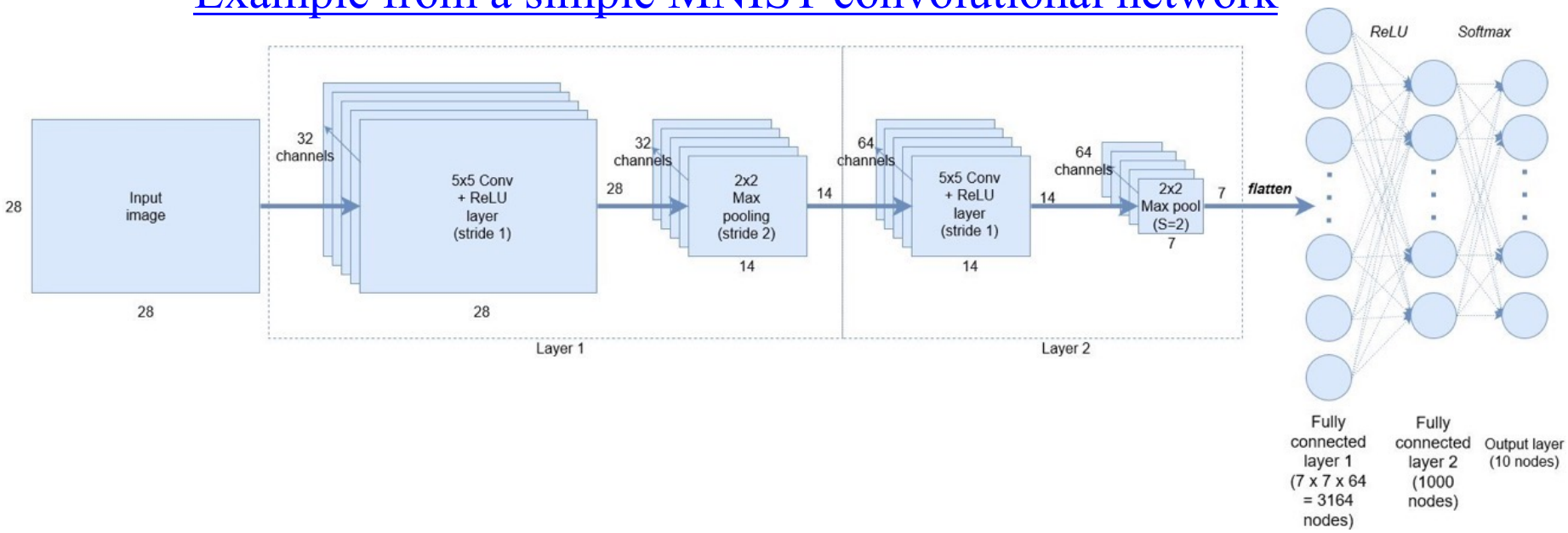
- Good for 2D image processing: classification, object recognition, automobile lane tracking, etc.
- Successive convolution layers learn higher-level features
- Classic demo: learn to recognize hand-written digits from MNIST data with 70K examples

In-Depth

Keras: API works with TensorFlow

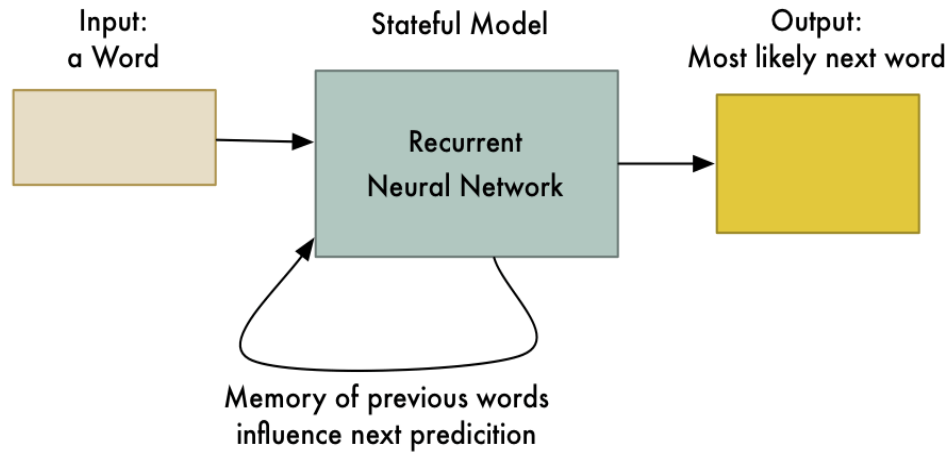
```
model = keras.Sequential(
    [
        keras.Input(shape=input_shape),
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dropout(0.5),
        layers.Dense(num_classes, activation="softmax"),
    ]
)
```

Example from a simple MNIST convolutional network



RNN: Recurrent Neural Networks

- Good for learning over sequences of data, e.g., a sentence of words
- LSTM: (Long Short Term Memory) a popular architecture that remembers and uses previous N inputs
- BI-LSTM: knows previous and upcoming inputs
- Attention: recent idea that learns long-range dependencies between inputs



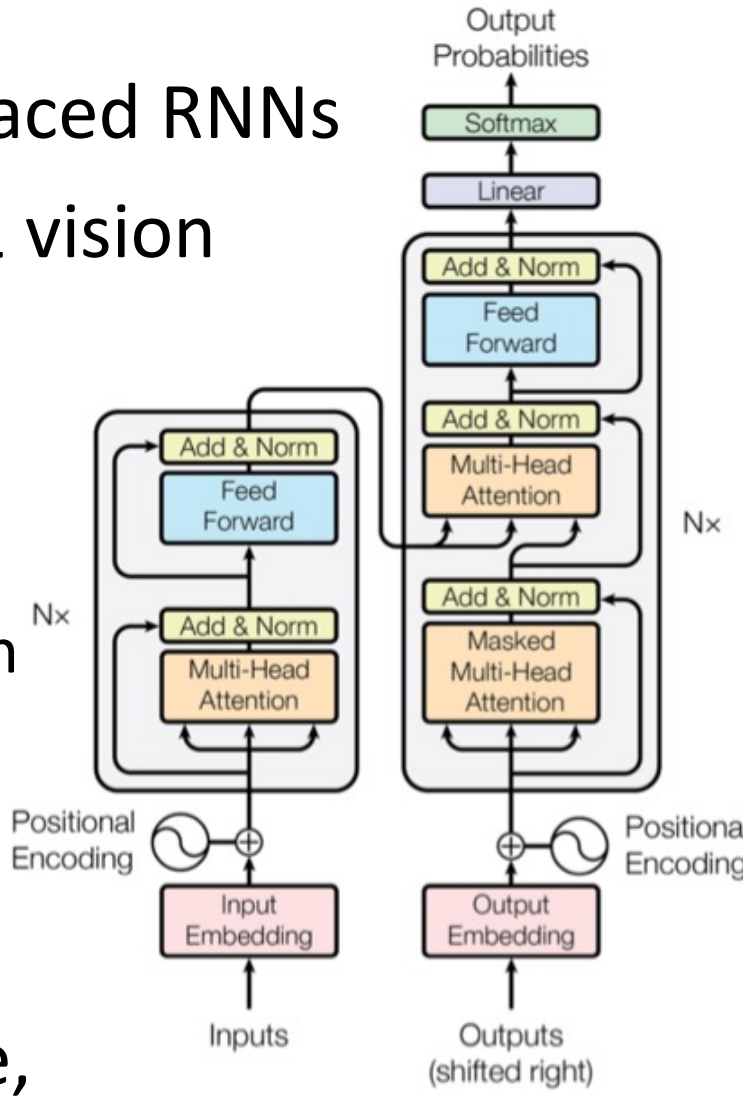
from Adam Geitgey

GAN: Generative Adversarial Network

- System of **two neural networks** competing against each other in a zero-sum game framework
- Provides a kind of **unsupervised learning** that improves the network
- Introduced by Ian Goodfellow et al. in 2014
- Can learn to draw samples from a model that is similar to data that we give them

Transformer

- Introduced in 2017 & has largely replaced RNNs
- Used primarily for natural language & vision processing tasks
- NLP applications ”transform” an input text into an output text
  - E.g., translation, summarization, question answering
- Uses encoder-decoder architecture with attention
- Popular pre-trainted models available, e.g. BERT and GPT

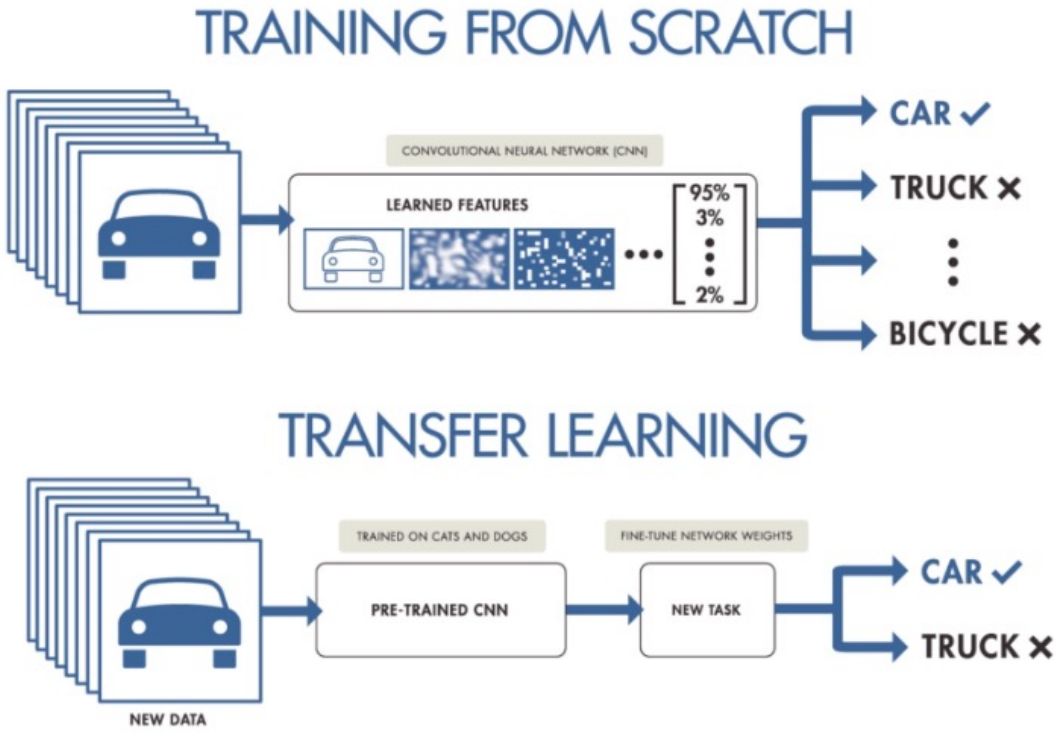


NNs Good at Transfer Learning

- Neural networks effective for transfer learning

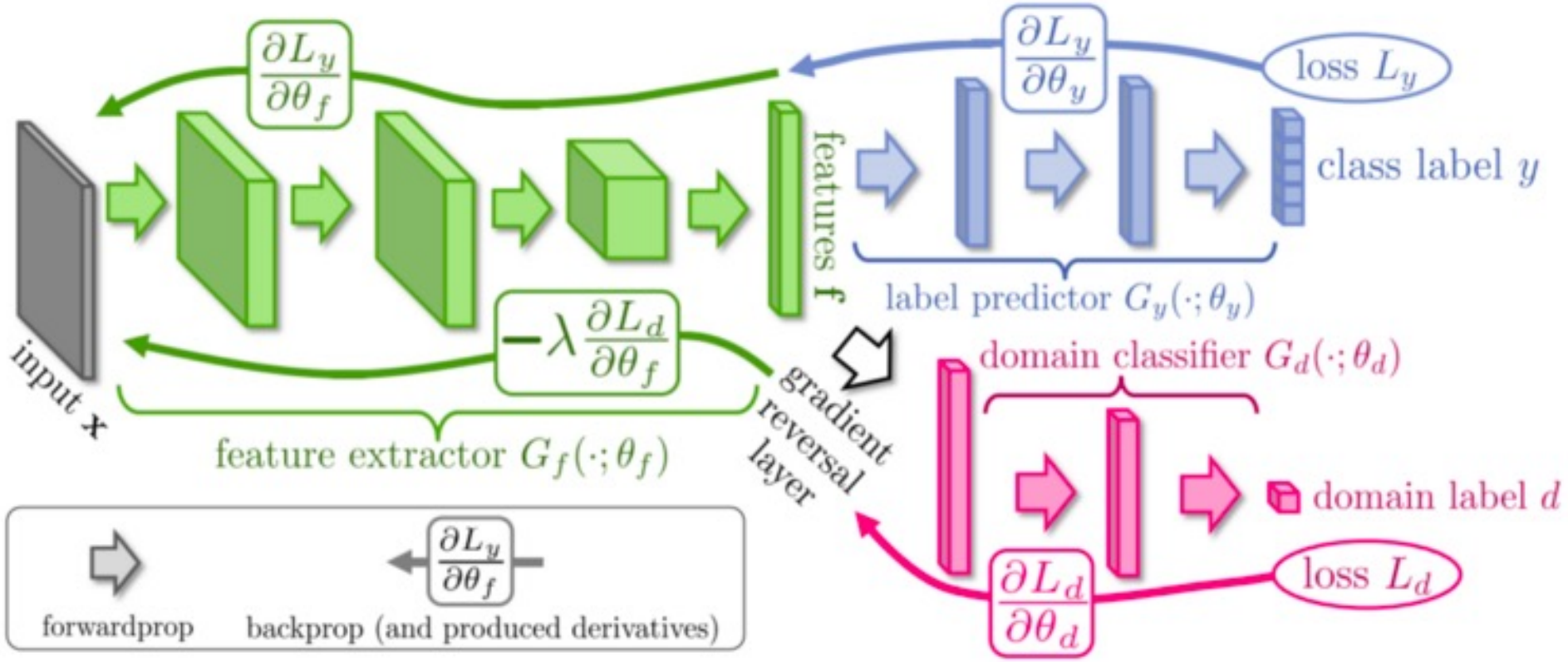
  Using parts of a model trained on a task as an initial model to train on a different task

- Particularly effective for image recognition and language understanding tasks



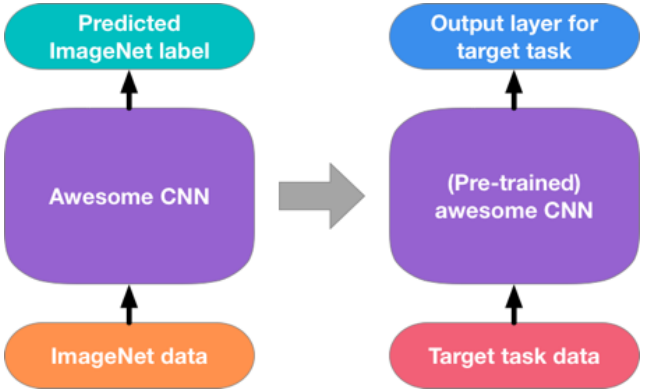
Good at Transfer Learning

- For images, the initial stages of a model learn high-level visual features (lines, edges) from pixels
- Final stages predict task-specific labels



source:http://ruder.io/transfer-learning/

Fine Tuning a NN Model



- Special kind of transfer learning
  - Start with a pre-trained model
  - Replace last output layer(s) with a new one(s)
  - One option: Fix all but last layer by marking as trainable:false
- Retraining on new task and data very fast
  - Only the weights for the last layer(s) are adjusted
- Example
  - Start: NN to classify animal pix with 100s of categories
  - Finetune on new task: classify pix of 10 common pets

Conclusions

- Quick intro to neural networks & deep learning
- Learn more by
  - Try scikit-learn's neural network models
  - Explore TensorFlow with Keras / PyTorch
  - Work through examples
- and then try your own project idea