Machine & Deep Learning

Pr Rahhal ERRATTAHI
rahhal.errattahi@um6p.ma
errattahi.r@ucd.ac.ma

Lecture 06

Neural Networks

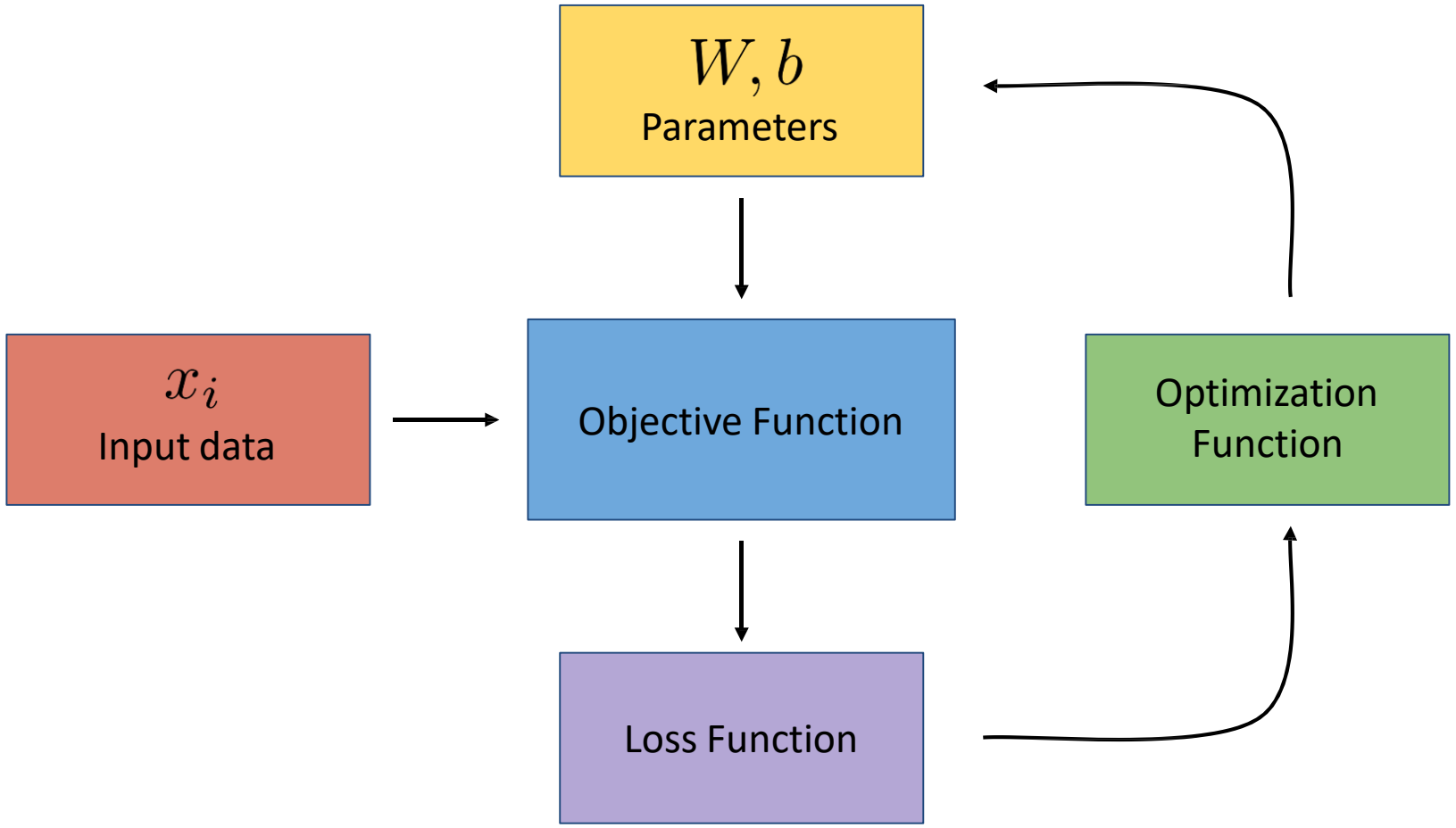
Lecture 6 Overview

- Neural architectures
- Training neural nets
- Neural network design
- Neural networks in practice
- Model selection
- Summary

Lecture 6 Overview

- Neural architectures
- Training neural nets
- Neural network design
- Neural networks in practice
- Model selection
- Summary

Overall Picture



Linear Classifier

$$f(x, W, b)$$

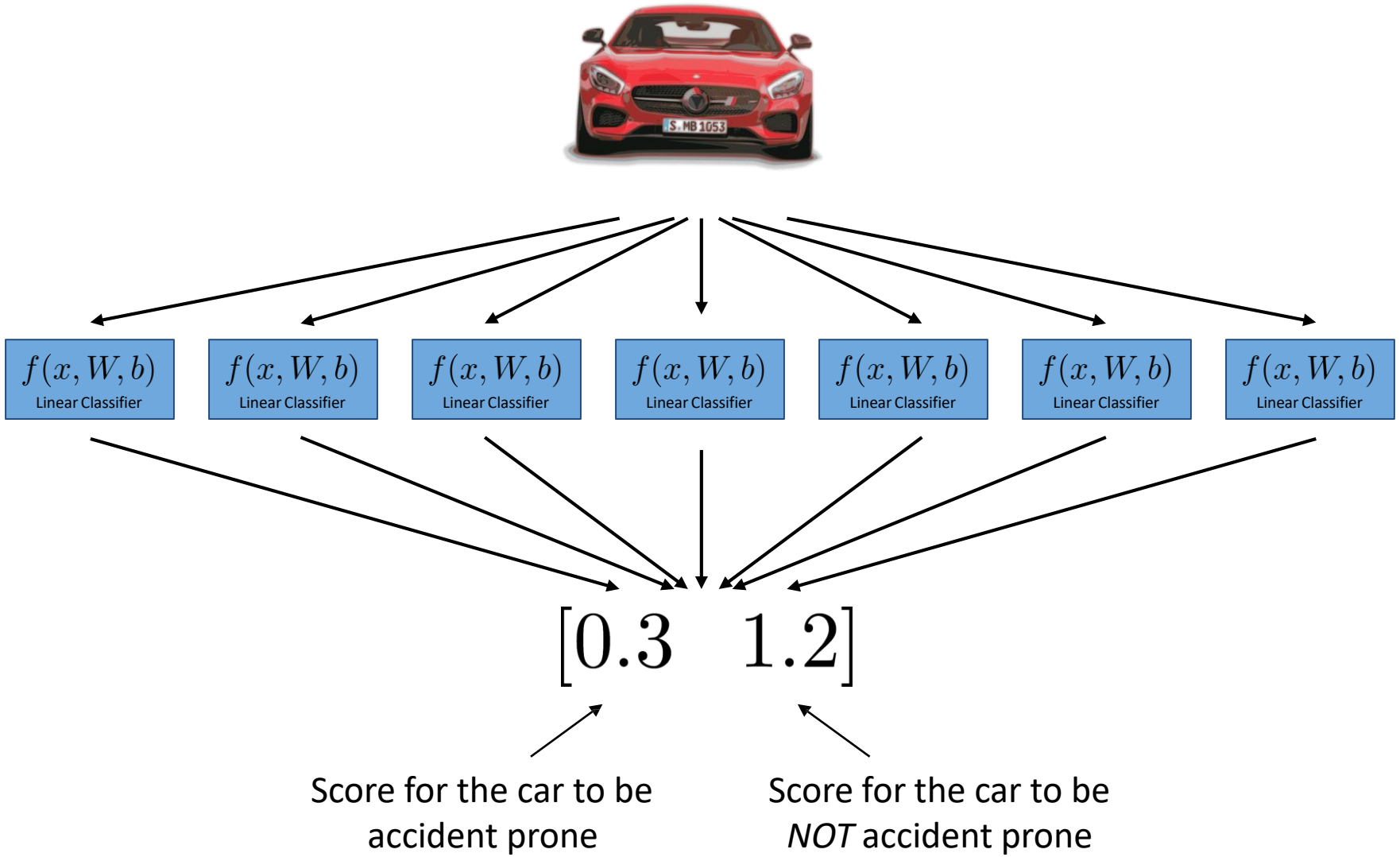
Linear Classifier

$$\begin{bmatrix} 0.3 & 1.2 \end{bmatrix}$$

Score for the car to be accident prone

Score for the car to be *NOT* accident prone

Neural Network



Neural Network

Color

Speed

Acceleration

Old cars

$f(x, W, b)$ Linear Classifier

$f(x, W, b)$ Linear Classifier

$f(x, W, b)$ Linear Classifier

$f(x, W, b)$ Linear Classifier

$f(x, W, b)$ Linear Classifier

$f(x, W, b)$ Linear Classifier

$f(x, W, b)$ Linear Classifier

As a simplification, we can say that, each classifier learns to look at a particular feature of the input (the *car* in this case)

Score for the car to be accident prone

Score for the car to be *NOT* accident prone

Neural Network

Color

Speed

Acceleration

Old cars

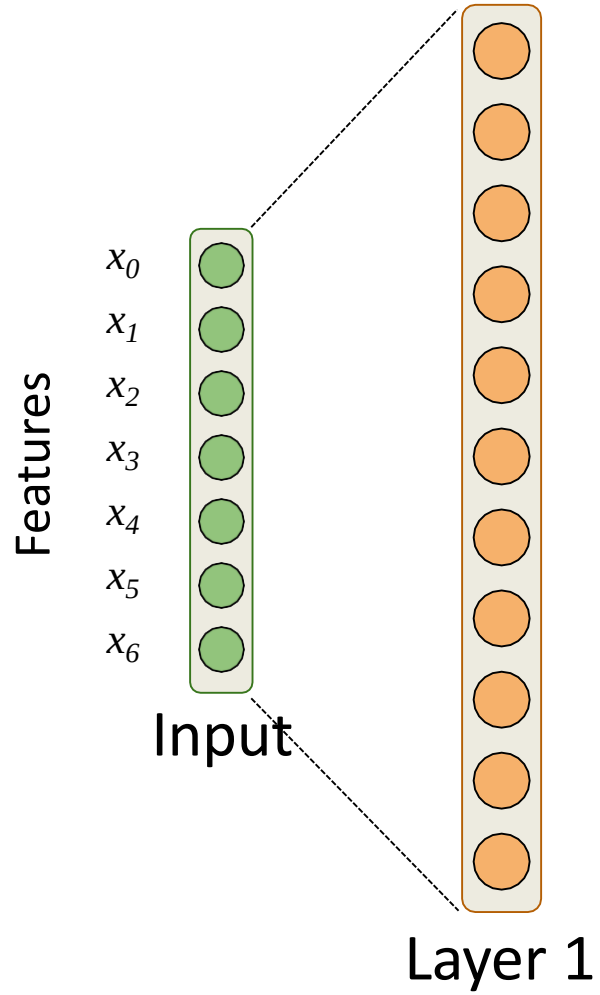
Neuron Neuron Neuron Neuron Neuron Neuron Neuron

As a simplification, we can say that, each **neuron** learns to look at a particular feature of the input (the *car* in this case)

Score for the car to be accident prone

Score for the car to be *NOT* accident prone

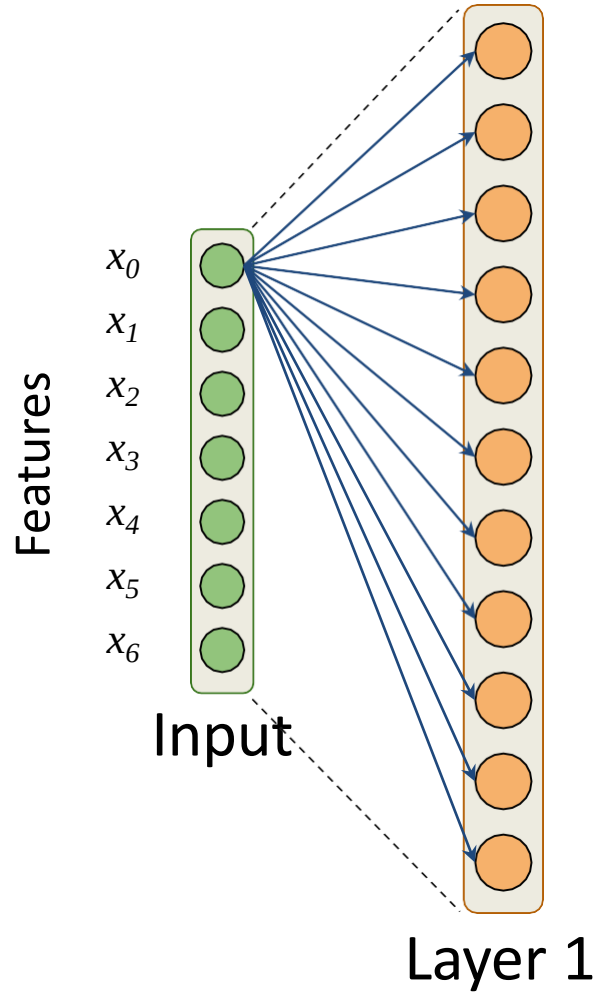
Neural Network



The neurons in the layer can be thought of as representing *richer features*

Think of these *richer features* as combinations of the *input features* we provided to the system

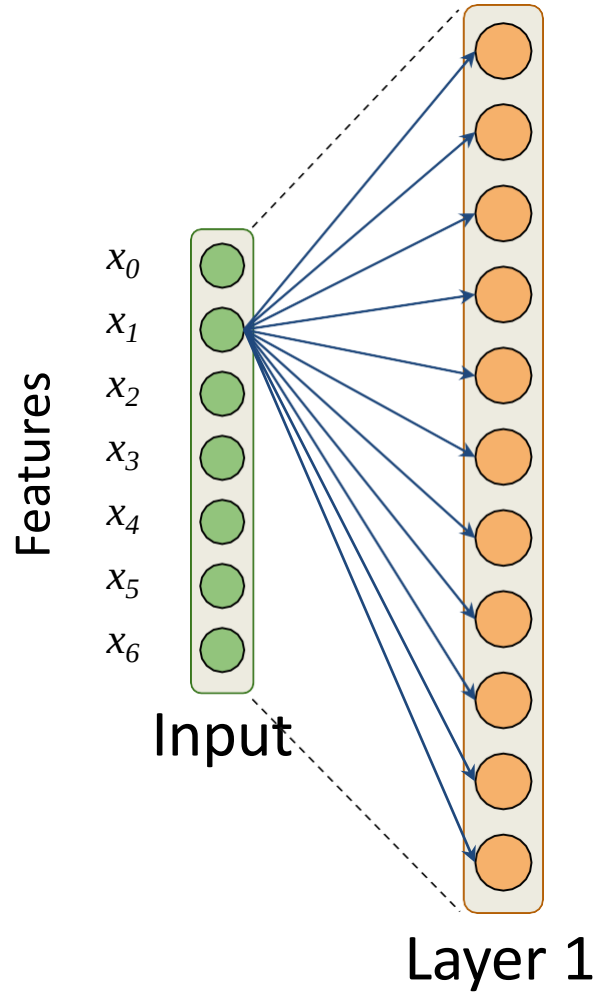
Neural Network



The neurons in the layer can be thought of as representing *richer features*

Think of these *richer features* as combinations of the *input features* we provided to the system

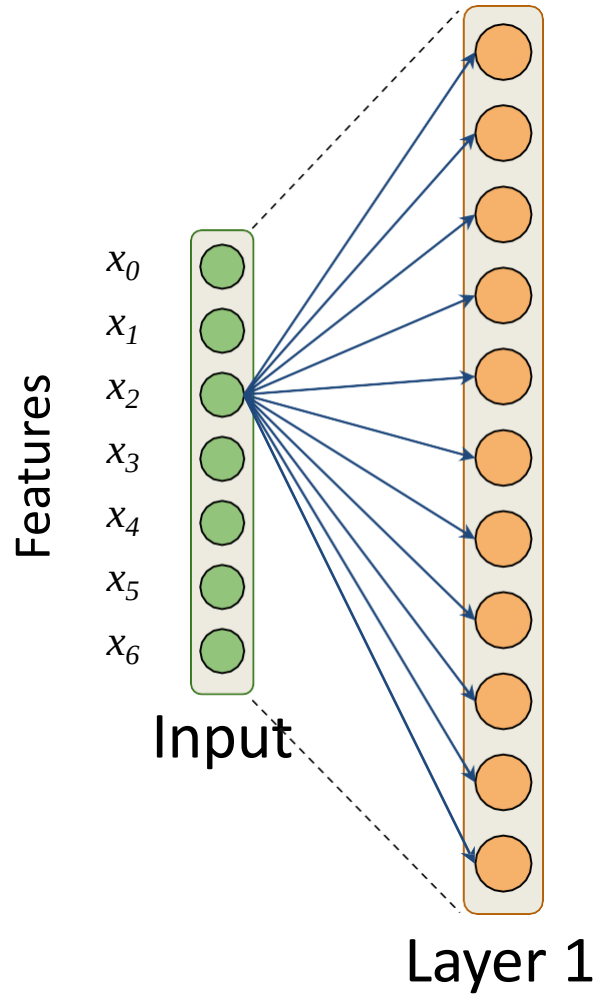
Neural Network



The neurons in the layer can be thought of as representing *richer features*

Think of these *richer features* as combinations of the *input features* we provided to the system

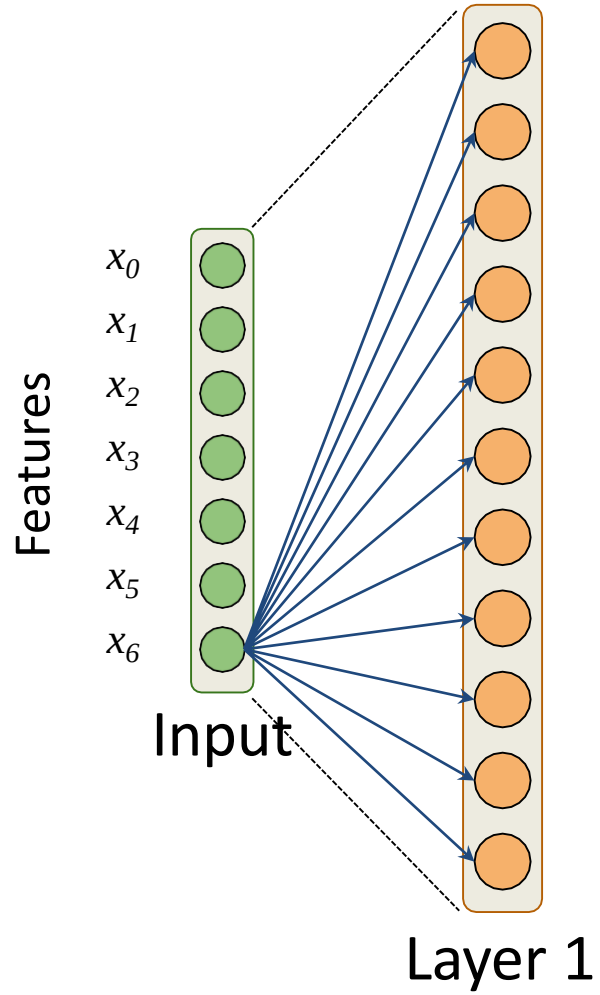
Neural Network



The neurons in the layer can be thought of as representing *richer features*

Think of these *richer features* as combinations of the *input features* we provided to the system

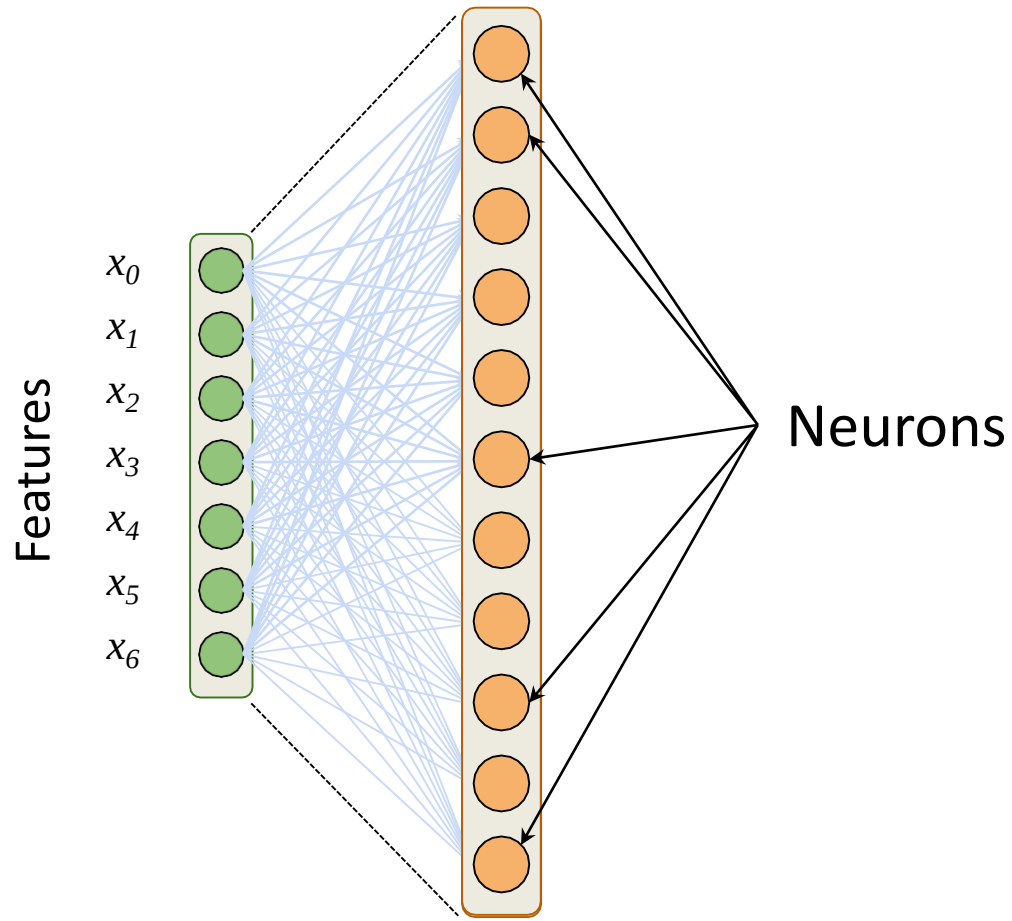
Neural Network



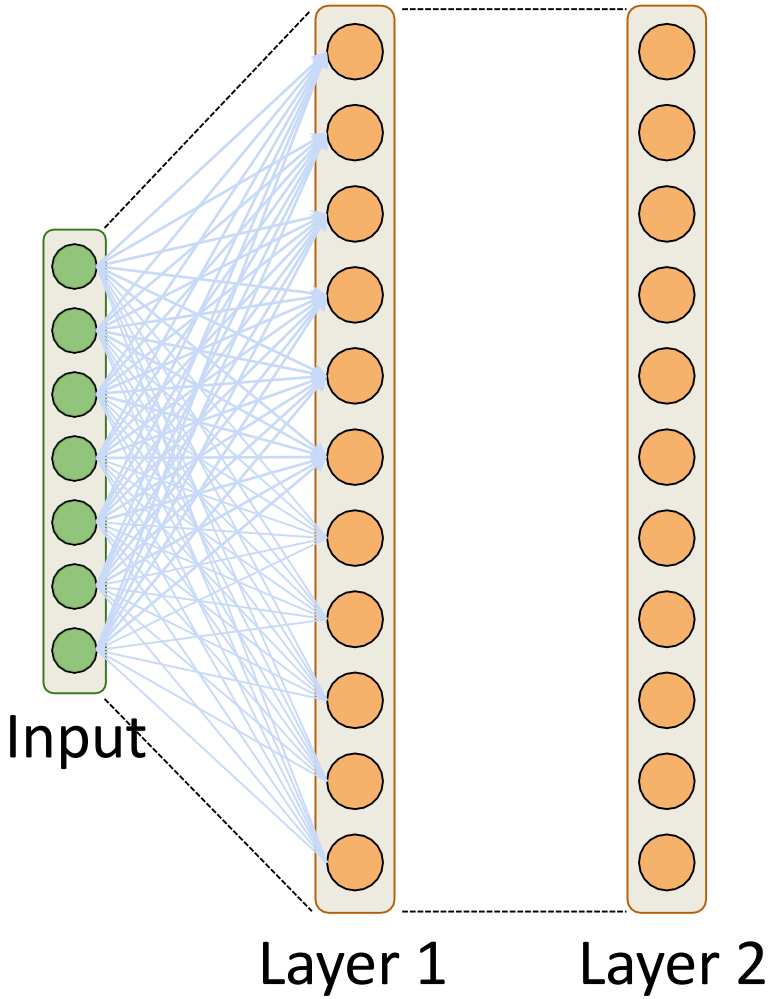
The neurons in the layer can be thought of as representing *richer features*

Think of these *richer features* as combinations of the *input features* we provided to the system

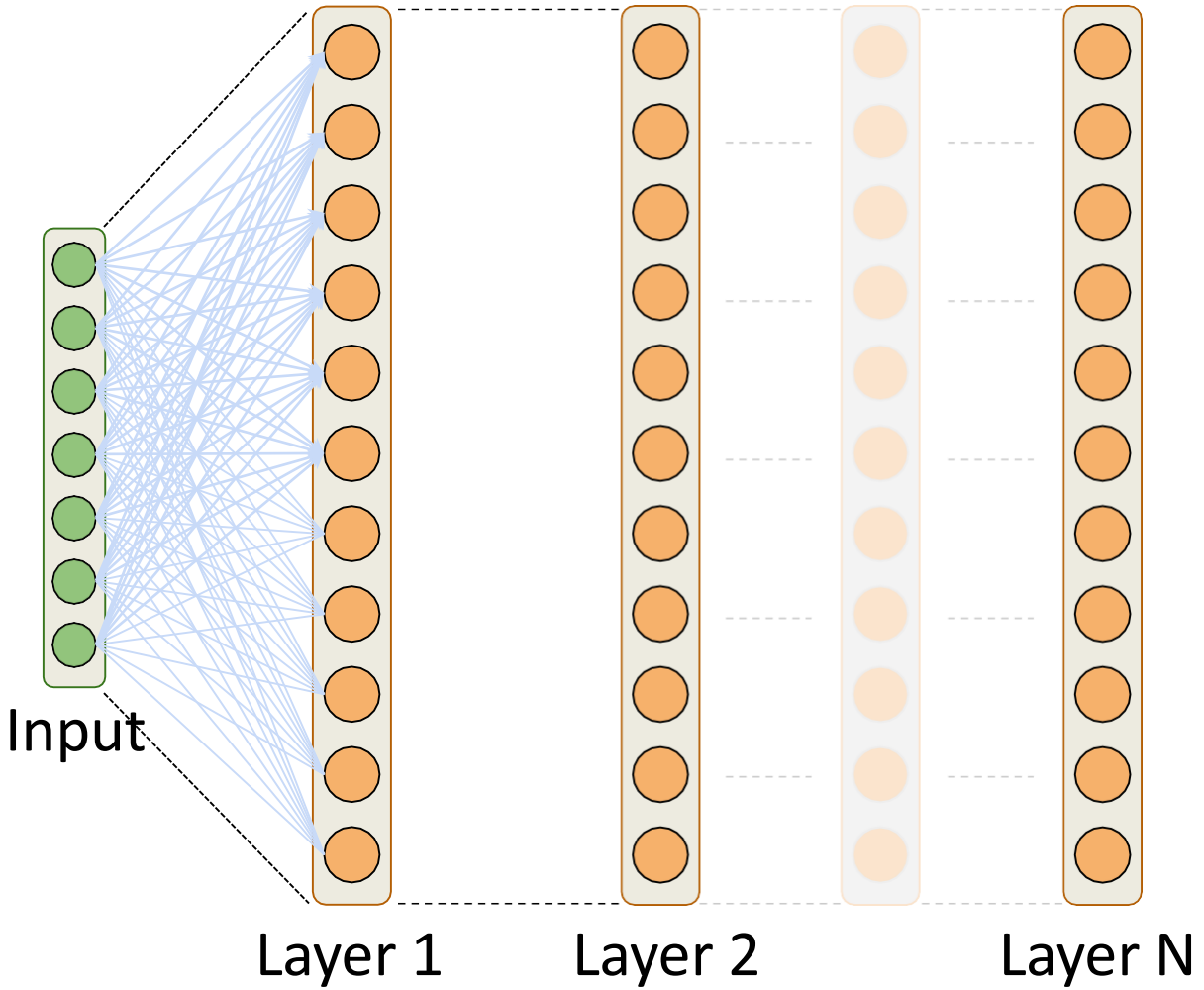
Neural Network



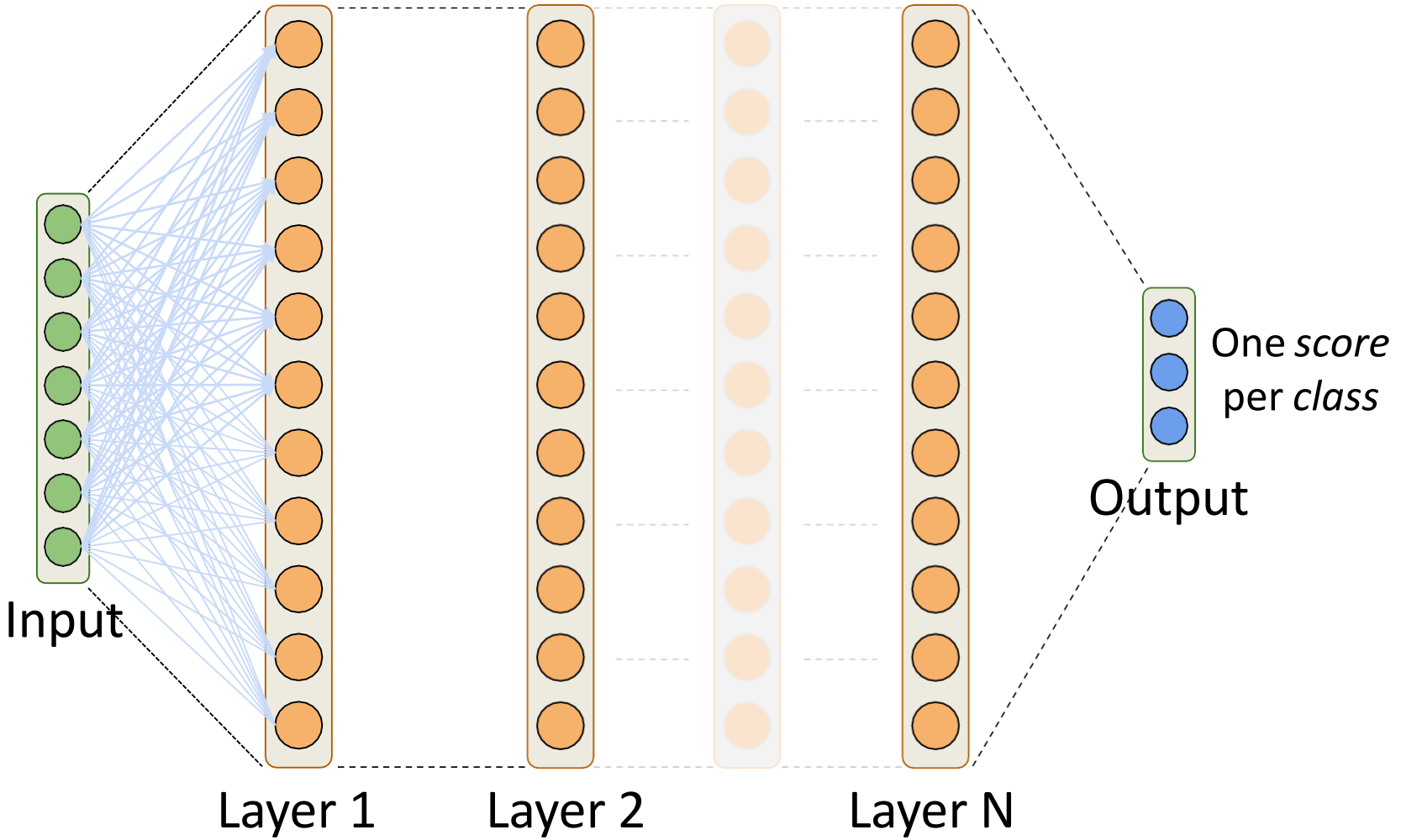
Neural Network



Neural Network

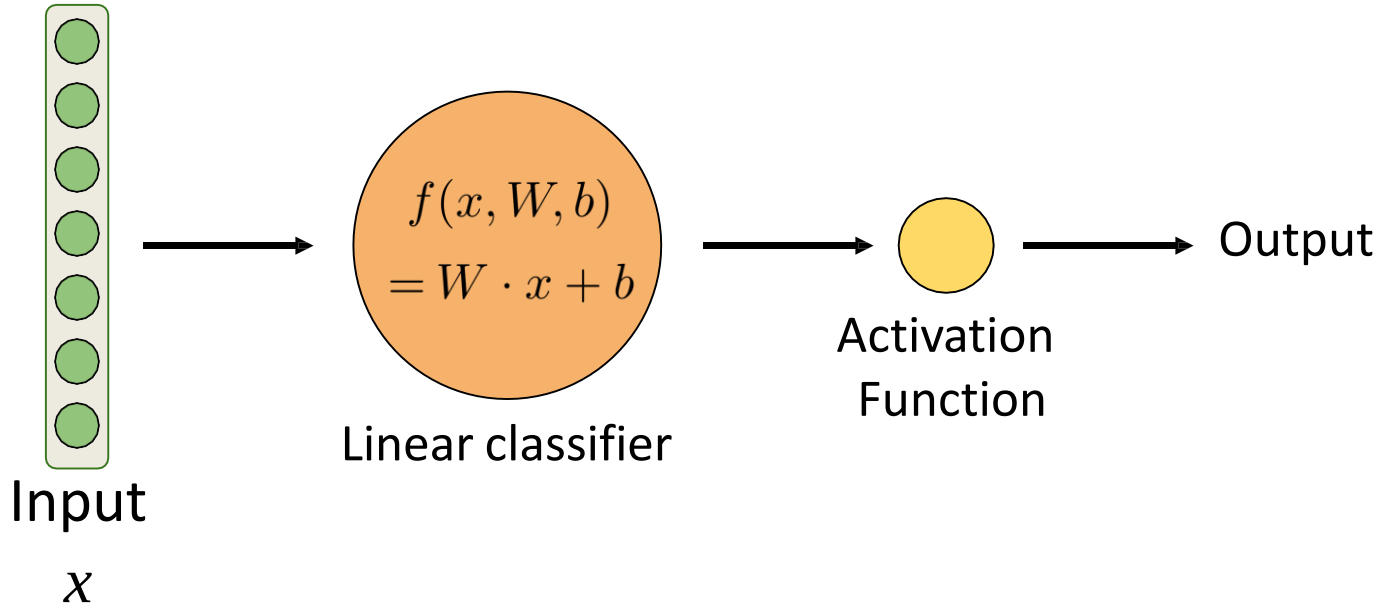


Neural Network



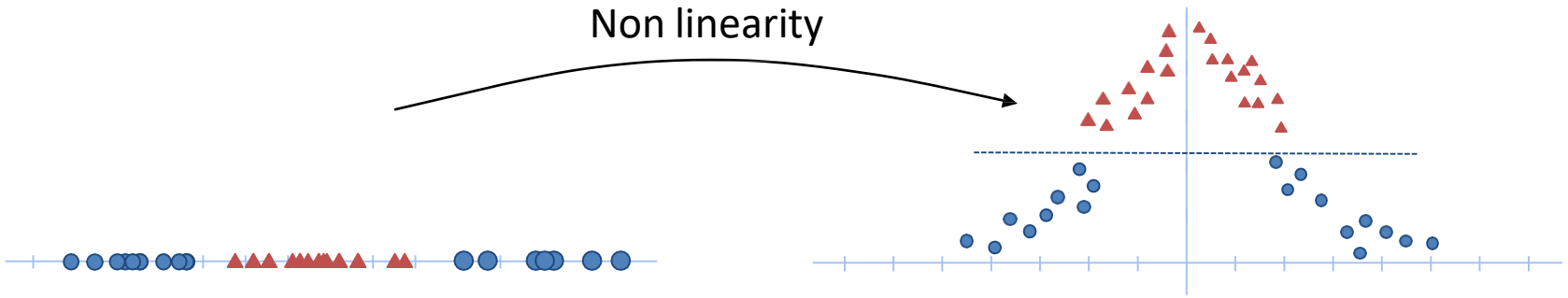
Neuron

A Neuron can be thought of as *a linear classifier* plus an *activation function*



Activation Functions

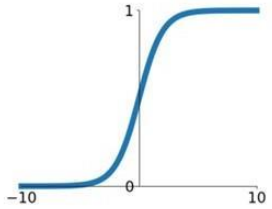
- Intuitively, a neuron looks at a particular feature of the data
- The activation after the linear classifier gives us an idea of how much the neuron “supports” the feature
- Activations also helps us map linear spaces into non- linear spaces



Activation Functions

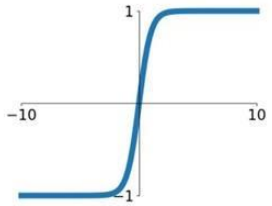
Sigmoid

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



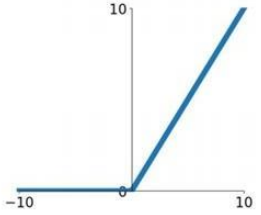
tanh

$$\tanh(x)$$



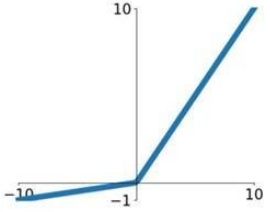
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

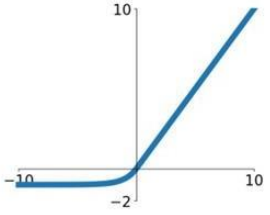


Maxout

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

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

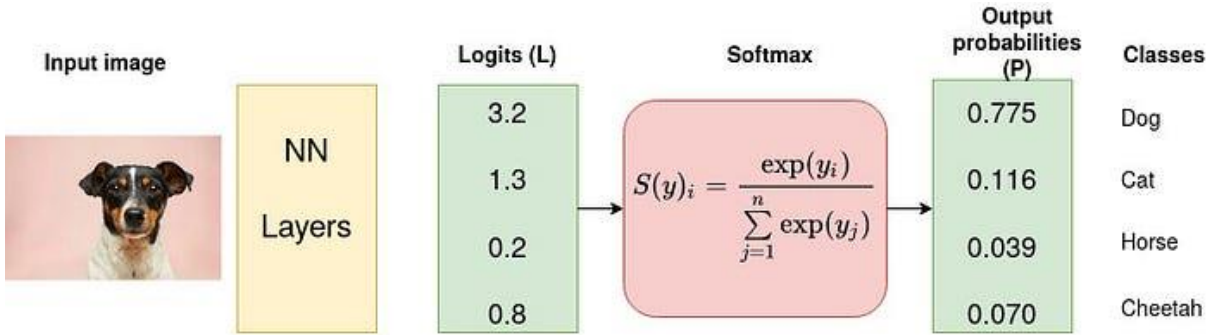


Activation functions for output layer

- *sigmoid* converts output to probability in [0,1]
  - For binary classification
- *softmax* converts all outputs (aka 'logits') to probabilities that sum up to 1
  - For multi-class classification (k classes)

$$softmax(x, i) = \frac{e^{x_i}}{\sum_{j=1}^{k} e^{x_j}}$$

- For regression, don't use any activation function, let the model learn the exact target

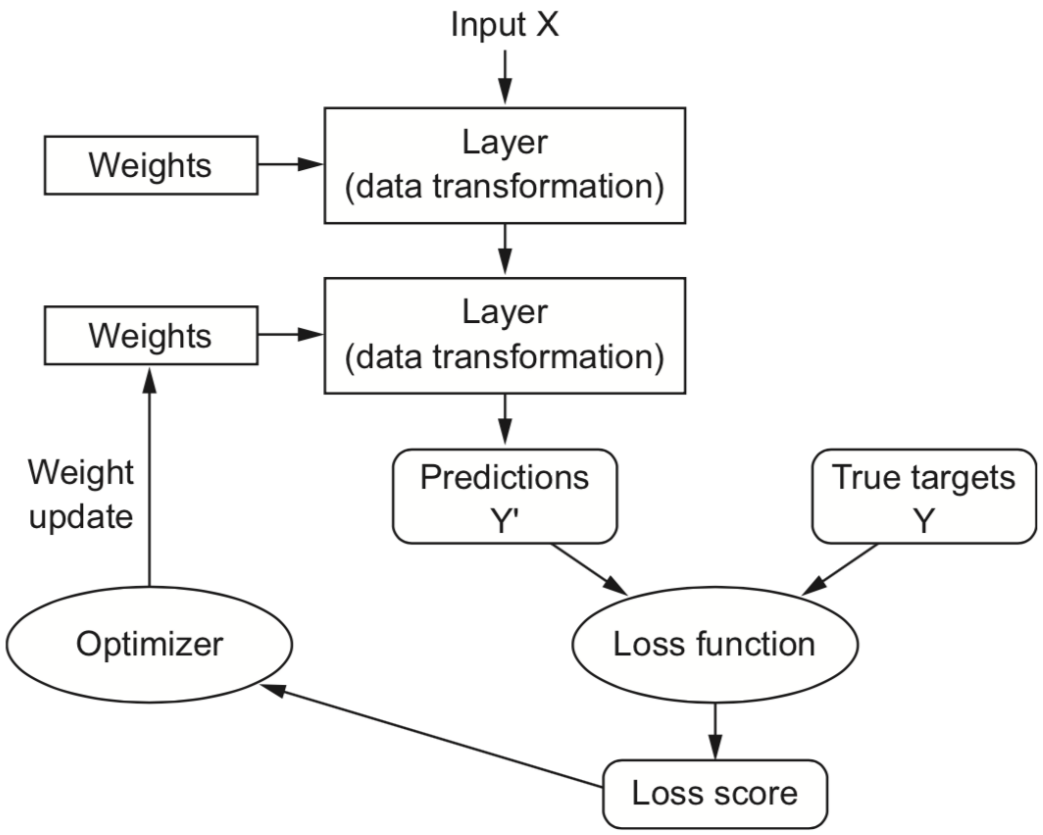


Lecture 6 Overview

- Neural architectures
- Training neural nets
- Neural network design
- Neural networks in practice
- Model selection
- Summary

Training Neural Nets

- Design the architecture, choose activation functions (e.g. sigmoids)
- Choose a way to initialize the weights (e.g. random initialization)
- Choose a *loss function* (e.g. log loss) to measure how well the model fits training data
- Choose an *optimizer* (typically an SGD variant) to update the weights



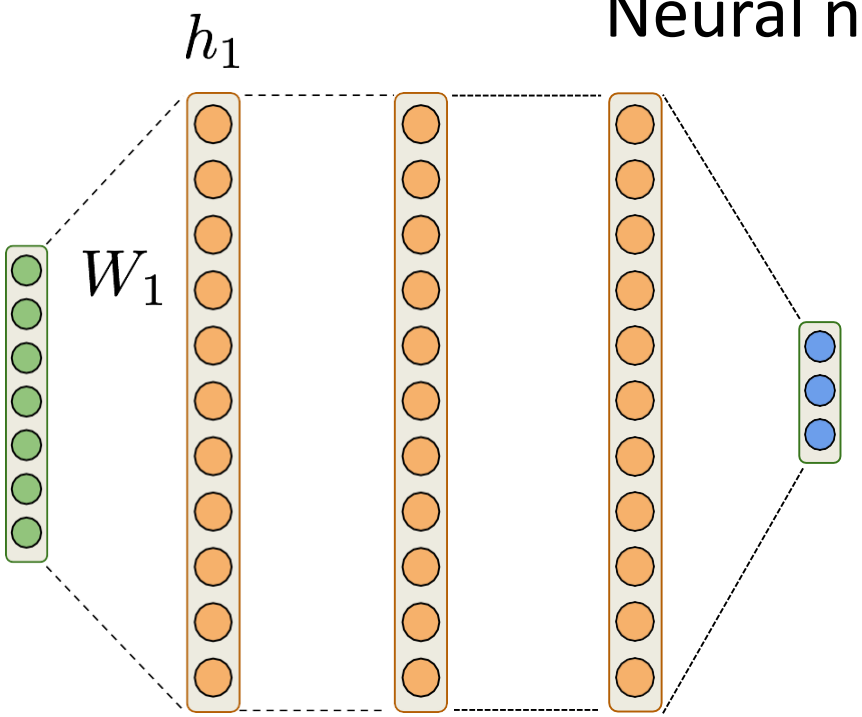
Training Neural Nets

- Entire network is nothing but a function:

$$f = W \cdot x + b$$

Linear classifier

Neural network with 3 hidden layers



$$h_1 = \sigma(W_1 \cdot x + b_1)$$

Activation function

Output of linear classifier "richer features"

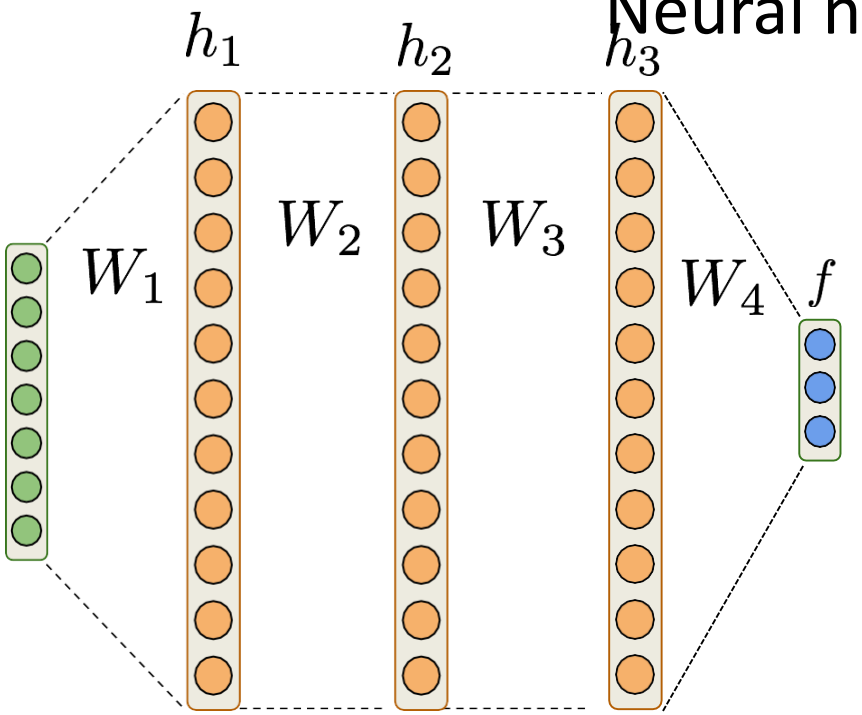
Training Neural Nets

- Entire network is nothing but a function:

$$f = W \cdot x + b$$

Linear classifier

Neural network with 3 hidden layers



$$h_1 = \sigma(W_1 \cdot x + b_1)$$

$$h_2 = \sigma(W_2 \cdot h_1 + b_2)$$

$$h_3 = \sigma(W_3 \cdot h_2 + b_3)$$

$$f = W_4 \cdot h_3 + b_4$$

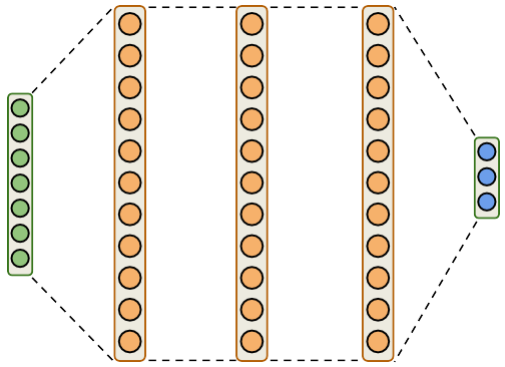
Final scores

Training Neural Nets

- Everything else remains the same!

$$f = W \cdot x + b$$

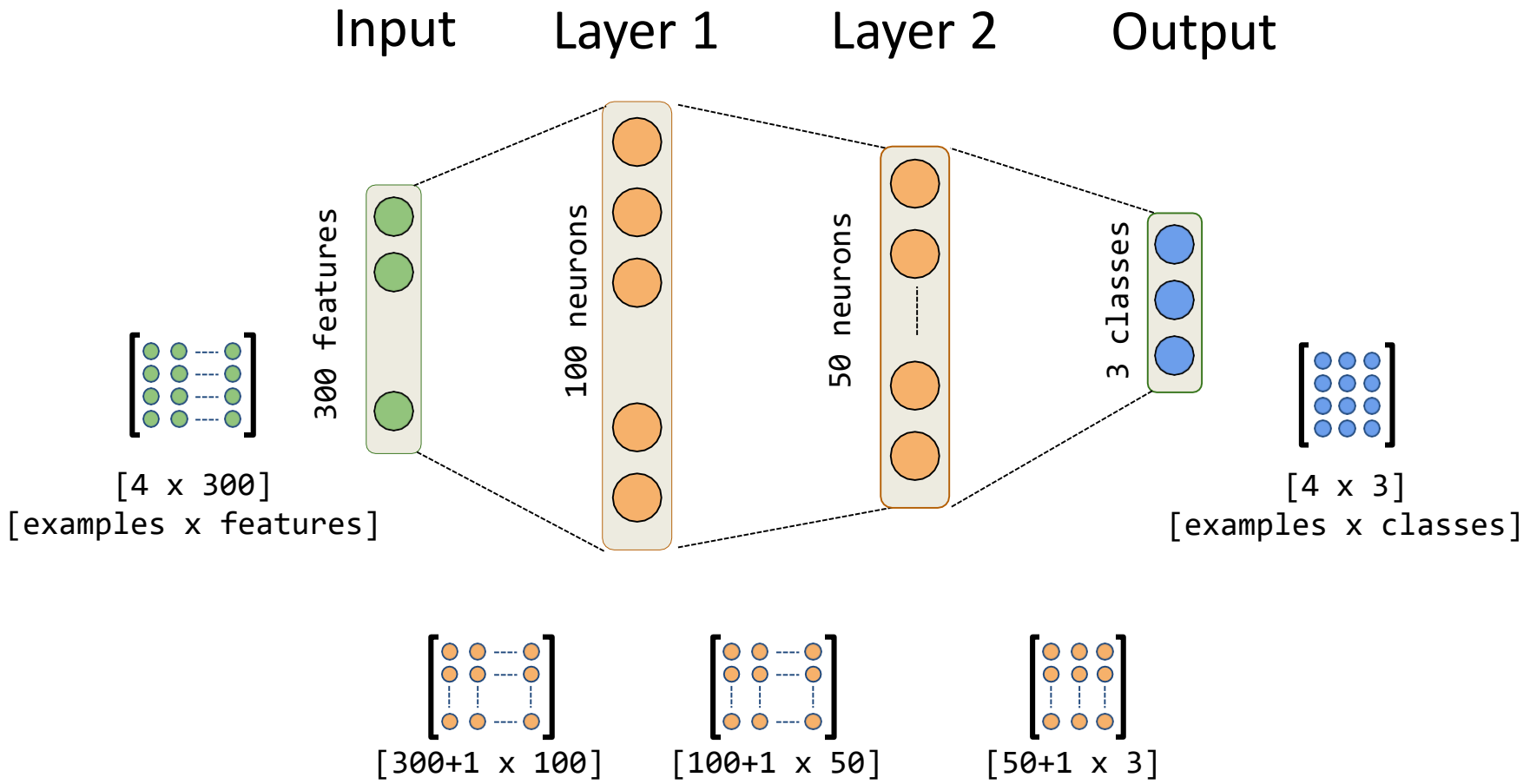
Linear classifier



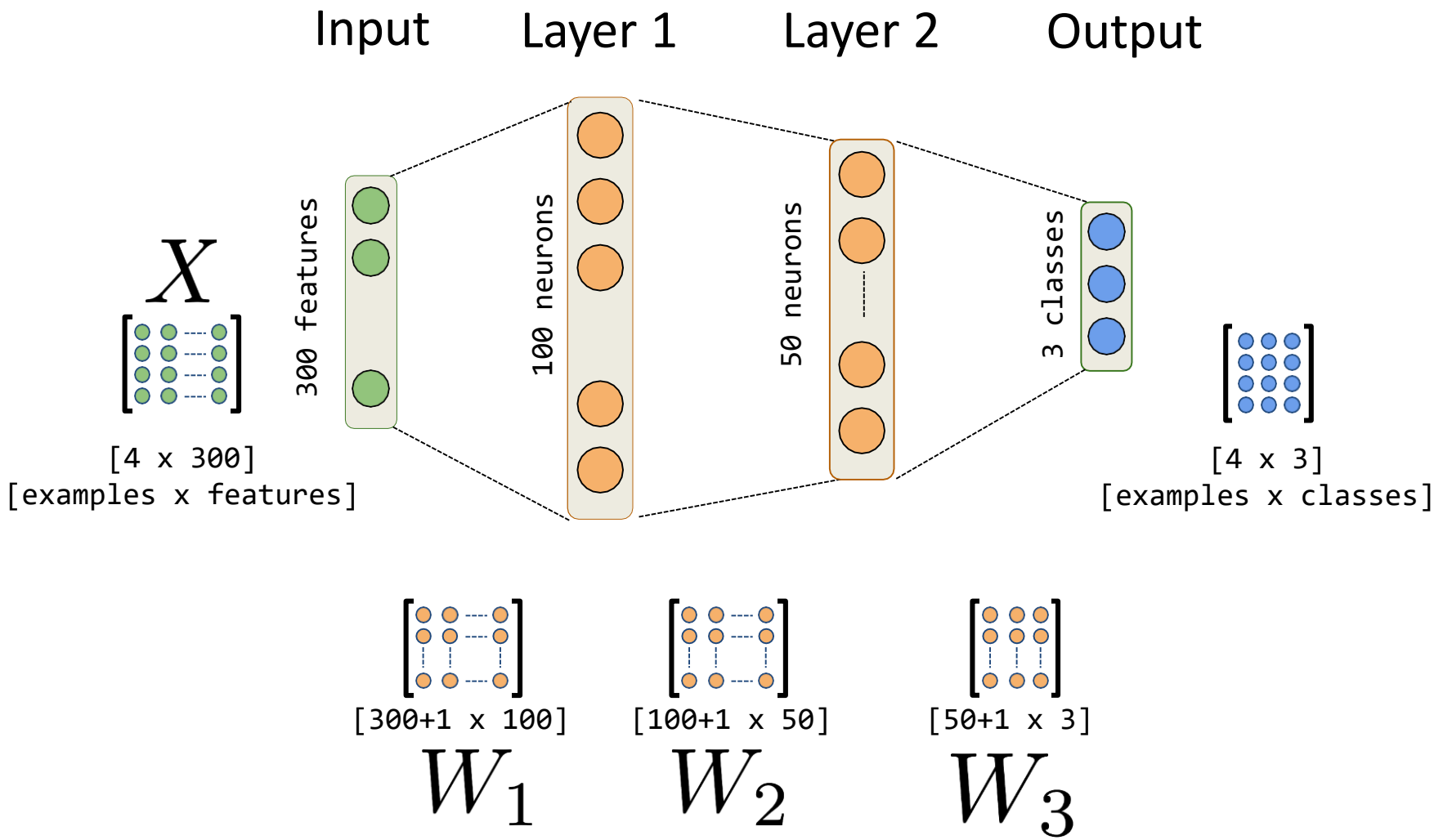
$$f = W_4 \cdot (\sigma(W_3 \cdot (\sigma(W_2 \cdot (\sigma(W_1 \cdot x + b_1)) + b_2)) + b_3)) + b_4$$

Neural network with 3 hidden layers

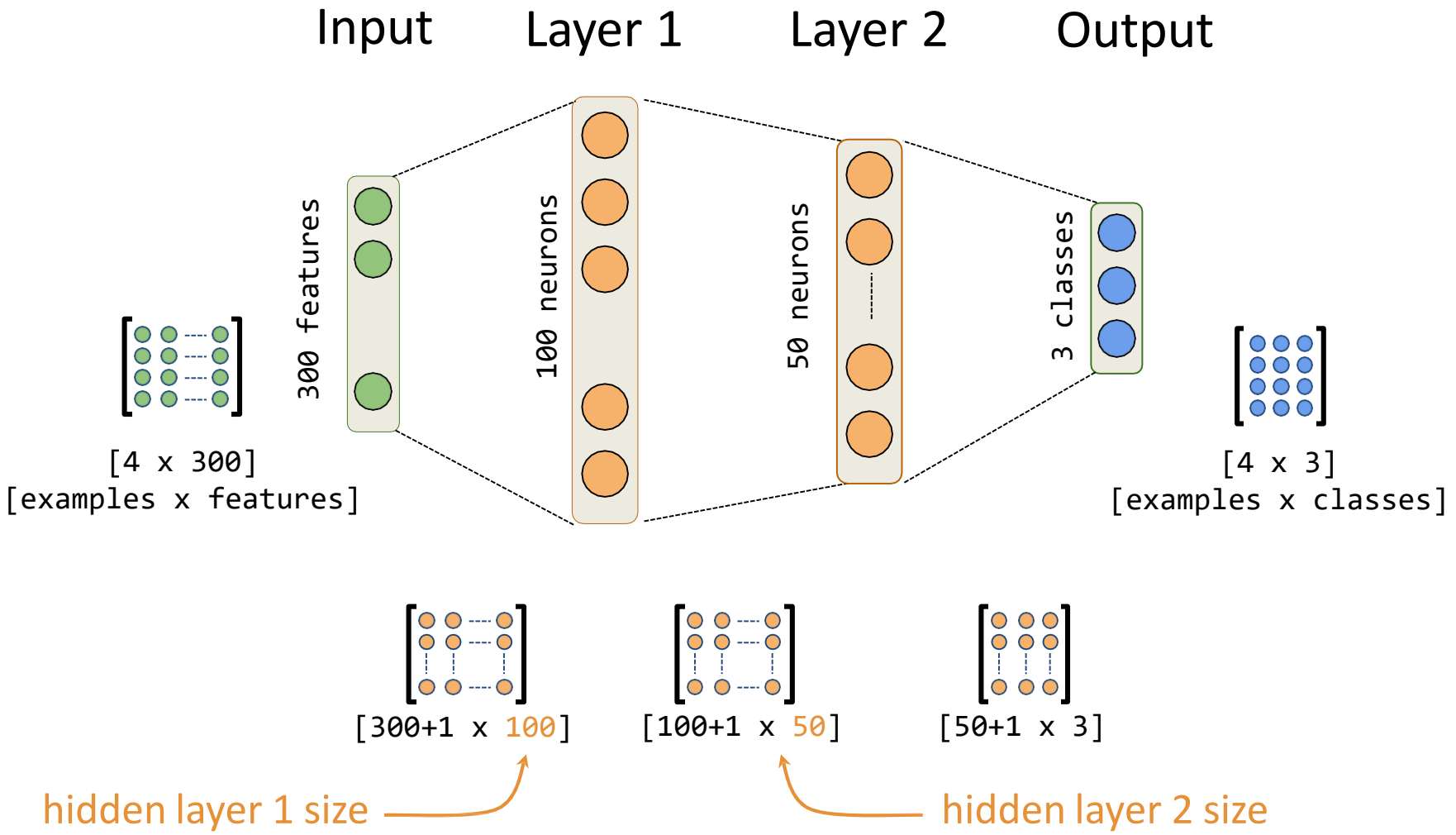
Training Neural Nets



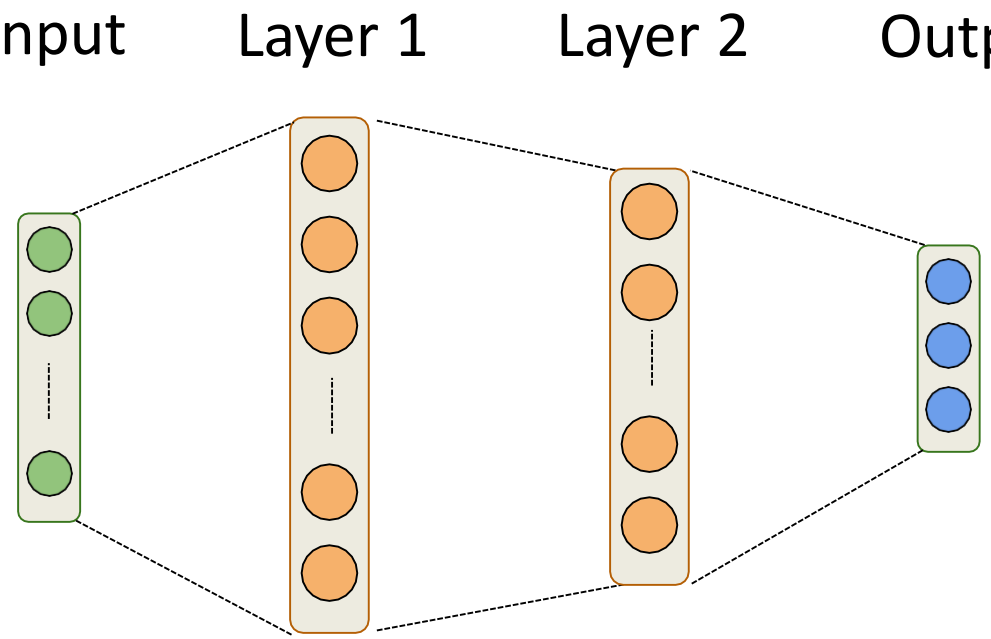
Training Neural Nets



Training Neural Nets



Training Neural Nets



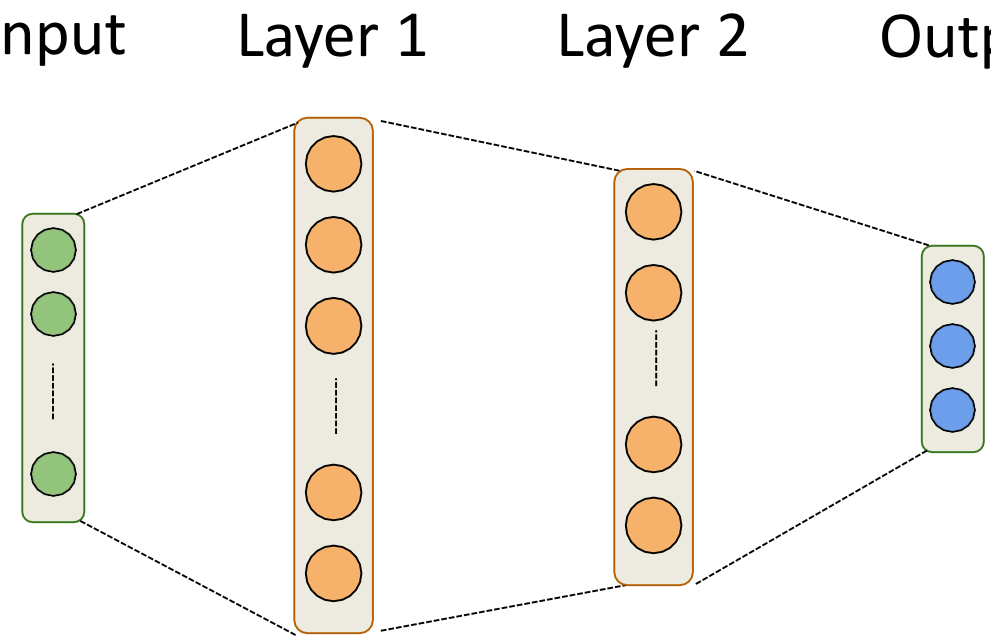
$$h_1 = \sigma(W_1 \cdot x + b_1)$$

$$h_2 = \sigma(W_2 \cdot h_1 + b_2)$$

$$f = W_3.h_2 + b_3$$

Forward Pass

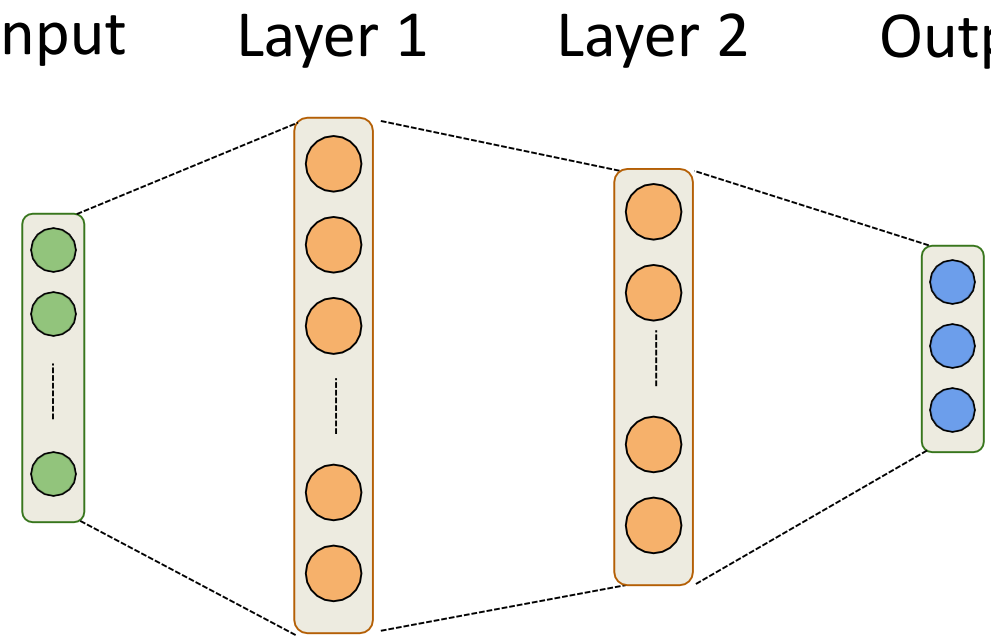
Training Neural Nets



$$L = -\log(f_c)$$

Cross Entropy Loss

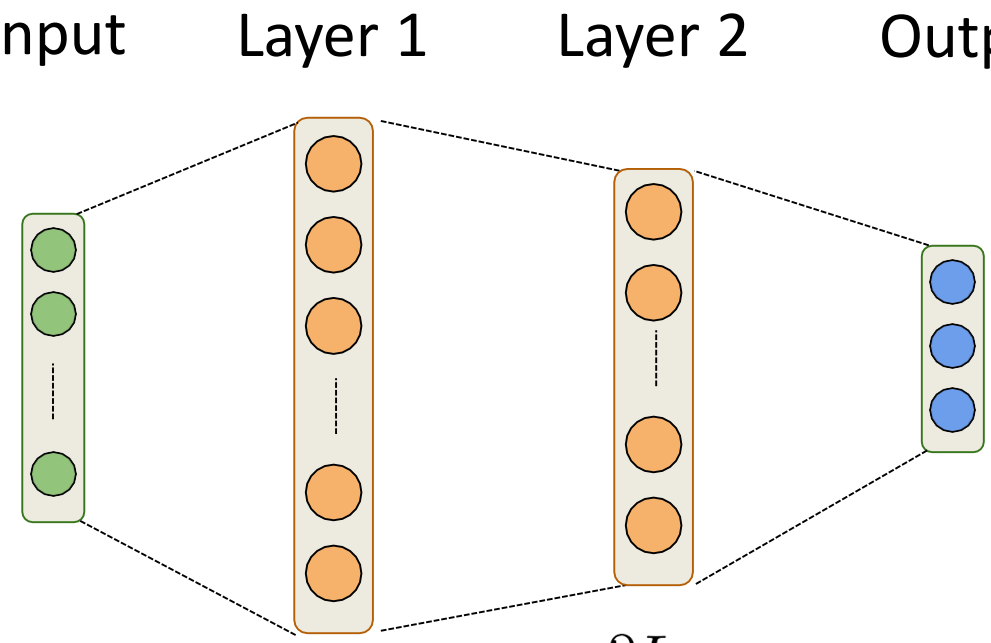
Training Neural Nets



Compute $\frac{\partial L}{\partial W_i}$ and $\frac{\partial L}{\partial b_i}$ using backpropagation

Optimization

Training Neural Nets



$$W_i = W_i - \eta \cdot \frac{\partial L}{\partial W_i}$$

$$b_i = b_i - \eta \cdot \frac{\partial L}{\partial b_i}$$

Backward Pass

Lecture 6 Overview

- Neural architectures
- Training neural nets
- Neural network design
- Neural networks in practice
- Model selection
- Summary

Weight initialization

- Initializing weights to 0 is bad: all gradients in layer will be identical (symmetry)
- Too small random weights shrink activations to 0 along the layers (vanishing gradient)
- Too large random weights multiply along layers (exploding gradient, zig-zagging)

Weight initialization

- Ideal: small random weights + variance of input and output gradients remains the same
  - Glorot/Xavier initialization (for tanh): randomly sample from

$$N(0, \sigma) = \sqrt{\frac{2}{\text{fan\_in+fan\_out}}}$$

    - fan_in: number of input units, fan_out: number of output units
  - He initialization (for ReLU): randomly sample from

$$N(0, \sigma) = \sqrt{\frac{2}{\text{fan\_out}}}$$

  - Uniform sampling (instead of $N(0, \sigma)$) for deeper networks (w.r.t. vanishing gradients)

Optimizers

- Using a constant learning η rate for weight updates

$$w_i = w_i - \eta \cdot \frac{\partial L}{\partial w_i}$$ is not ideal

  - You would need to ‘magically’ know the right value

Optimizers

SGD with learning rate schedules:

- Learning rate decay/annealing with decay rate k
  - E.g. exponential ($\eta_{s-1} = \eta_0 e^{-ks}$), inverse-time ($\eta_{s-1} = \frac{\eta_0}{1+ks}$),…

Momentum:

- Adds a velocity vector **v** with momentum γ (e.g. 0.9, or increase from $\gamma = 0.5$ to $\gamma = 0.9$)

$$w_{s+1} = w_s + v_s \text{ with } v_s = \gamma v_{s-1} - \eta \cdot \frac{\partial L}{\partial w_s}$$

Optimizers

Adagrad:

- scale η according to squared sum of previous gradients $G_{i,s} = \sum_{t=1}^{s} \left(\frac{\partial L}{\partial w_{i,t}} \right)^2$
- Update rule for w_i . Usually $\varepsilon = 10^{-7}$ (avoids division by 0),$\eta = 0.001$.

$$w_{i,s+1} = w_{i,s} - \frac{\eta}{G_{i,s} + \varepsilon} \cdot \frac{\partial L}{\partial w_i}$$

Optimizers

RMSProp:

- use *moving average* of squared gradients $m_{i,s} = \gamma m_{i,s-1} + (1-\gamma)\left(\frac{\partial L}{\partial w_i}\right)^2$
- Avoids that gradients dwindle to 0 as $m_{i,s}$ grows. Usually $\gamma = 0.9$, η =0.001

$$w_{i,s+1} = w_{i,s} - \frac{\eta}{\sqrt{m_{i,s} + \varepsilon}} \cdot \frac{\partial L}{\partial w_i}$$

Adam:

- RMSProp + momentum

Lecture 6 Overview

- Neural architectures
- Training neural nets
- Neural network design
- Neural networks in practice
- Model selection
- Summary

Neural networks in practice

```
network = models.Sequential()
network.add(layers.Dense(512, activation='relu', kernel_initializer='he_normal', input_shape=(28 * 28,)))
network.add(layers.Dense(512, activation='relu', kernel_initializer='he_normal'))
network.add(layers.Dense(10, activation='softmax'))
```

- Input layer (‘input_shape’): a flat vector of 28*28=784 nodes
  - We’ll see how to properly deal with images later
- Two dense hidden layers: 512 nodes each, ReLU activation
  - Glorot weight initialization is applied by default
- Output layer: 10 nodes (for 10 classes) and softmax activation

Neural networks in practice

```
network.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 512)               401920

 dense_1 (Dense)             (None, 512)               262656

 dense_2 (Dense)             (None, 10)                5130

=================================================================
Total params: 669706 (2.55 MB)
Trainable params: 669706 (2.55 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

- Lots of parameters (weights and biases) to learn!
  - hidden layer 1 : (28 * 28 + 1) * 512 = 401920
  - hidden layer 2 : (512 + 1) * 512 = 262656
  - output layer: (512 + 1) * 10 = 5130

Neural networks in practice

- **Loss function**
  - Cross-entropy (log loss) for multi-class classification (y_{true} is one-hot encoded)
  - Use binary crossentropy for binary problems (single output node)
  - Use sparse categorical crossentropy if y_{true} is label-encoded (1,2,3,…)
- **Optimizer**
  - Any of the optimizers we discussed before. RMSprop usually works well.
- **Metrics**
  - To monitor performance during training and testing, e.g. accuracy

```
# Shorthand
network.compile(loss='categorical_crossentropy', optimizer='rmsprop', metrics=['accuracy'])
# Detailed
network.compile(loss=CategoricalCrossentropy(label_smoothing=0.01), optimizer=RMSprop(learning_rate=0.001, momentum=0.0) metrics=[Accuracy()])
```

Neural networks in practice

- Always normalize (standardize or min-max) the inputs. Mean should be close to 0.
  - Avoid that some inputs overpower others
  - Speed up convergence
- Reshape the data to fit the shape of the input layer, e.g. (n, 28*28) or (n, 28,28)
  - Tensor with instances in first dimension, rest must match the input layer
- In multi-class classification, every class is an output node, so one-hot-encode the labels
  - e.g. class ‘4’ becomes [0,0,0,0,1,0,0,0,0,0]

```
X = X.astype('float32') / 255
X = X.reshape((60000, 28 * 28))
y = to_categorical(y)
```

Neural networks in practice

- Number of epochs: enough to allow convergence
  - Too much: model starts overfitting (or just wastes time)
- Batch size: small batches (e.g. 32, 64,… samples) often preferred
  - ‘Noisy’ training data makes overfitting less likely
    - Larger batches generalize less well (‘generalization gap’)
  - Requires less memory (especially in GPUs)
  - Large batches do speed up training, may converge in fewer epochs

```
history = network.fit(X_train, y_train, epochs=3, batch_size=32);
```

Neural networks in practice

- We can now call *predict* to generate predictions, and evaluate the trained model on the entire test set

```
network.predict(X_test) test_loss,
test_acc = network.evaluate(X_test, y_test)
print ('Test accuracy:', test_acc)


Test accuracy: 0.7547000050544739
```

Neural networks in practice

Let’s see it in action!

Demo - Neural Network with Spiral Data

Lecture 6 Overview

- Neural architectures
- Training neural nets
- Neural network design:
- Neural networks in practice
- Model selection
- Summary

Model selection

- How many epochs do we need for training?
- Train the neural net and track the loss after every iteration on a validation set
  - You can add a callback to the fit version to get info on every epoch
- Best model after a few epochs, then starts overfitting

Model selection

Early stopping

- Stop training when the validation loss (or validation accuracy) no longer improves
- Loss can be bumpy: use a moving average or wait for k steps without improvement

```
earlystop = callbacks.EarlyStopping(monitor='val_loss', patience=3)
model.fit(x_train, y_train, epochs=25, batch_size=512, callbacks=[earlystop])
```

Model selection

Regularization and memorization capacity

- The number of learnable parameters is called the model *capacity*
- A model with more parameters has a higher *memorization capacity*
  - Too high capacity causes overfitting, too low causes underfitting
  - In the extreme, the training set can be ‘memorized’ in the weights
- Smaller models are forced it to learn a compressed representation that generalizes better
  - Start with few parameters, increase until overfitting stars.

Model selection

Weight regularization (weight decay)

- As we did many times before, we can also add weight regularization to our loss function
- L1 regularization: leads to *sparse networks* with many weights that are 0
- L2 regularization: leads to many very small weights

```
network = models.Sequential() network.add(layers.Dense(256, activation='relu',
kernel_regularizer=regularizers.l2(0.001), input_shape=(28 * 28,))) network.add(layers.Dense(128, activation='relu',
kernel_regularizer=regularizers.l2(0.001)))
```

Model selection

Dropout

- Every iteration, randomly set a number of activations a_i to 0
- *Dropout rate* : fraction of the outputs that are zeroed-out (e.g. 0.1 - 0.5)
- Idea: break up accidental non-significant learned patterns
- At test time, nothing is dropped out, but the output values are scaled down by the dropout rate
  - Balances out that more units are active than during training

Model selection

Dropout

- Dropout is usually implemented as a special layer

```
network = models.Sequential()
network.add(layers.Dense(256, activation='relu', input_shape=(28 * 28,)))
network.add(layers.Dropout(0.5))
network.add(layers.Dense(32, activation='relu'))
network.add(layers.Dropout(0.5))
network.add(layers.Dense(10, activation='softmax'))
```

Model selection

Batch Normalization

- Batch normalization: normalize the activations of the previous layer within each batch
  - Within a batch, set the mean activation close to 0 and the standard deviation close to 1
  - Allows deeper networks less prone to vanishing or exploding gradients

Model selection

Batch Normalization

```
network = models.Sequential()
network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
network.add(layers.BatchNormalization())
network.add(layers.Dropout(0.5))
network.add(layers.Dense(256, activation='relu'))
network.add(layers.BatchNormalization())
network.add(layers.Dropout(0.5))
network.add(layers.Dense(64, activation='relu'))
network.add(layers.BatchNormalization())
network.add(layers.Dropout(0.5))
network.add(layers.Dense(32, activation='relu'))
network.add(layers.BatchNormalization())
network.add(layers.Dropout(0.5))
```

Lecture 6 Overview

- Neural architectures
- Training neural nets
- Neural network design:
- Neural networks in practice
- Model selection
- Summary

Summary

- Neural architectures
- Training neural nets
  - Forward pass: Tensor operations
  - Backward pass: Backpropagation
- Neural network design:
  - Activation functions
  - Weight initialization
  - Optimizers
- Neural networks in practice
- Model selection
  - Early stopping
  - Memorization capacity and information bottleneck
  - L1/L2 regularization
  - Dropout
  - Batch normalization

Lab 8 - Neural networks