BST 261: Data Science II
Lecture 2
MLPs

Heather Mattie
Harvard T.H. Chan School of Public Health
Spring 2 2020

Recipe of the Day

Blueberry scones with lemon glaze

© Sally's Baking Addiction

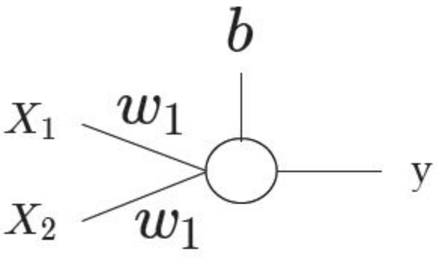
Deep learning glossaries

1. Google
2. WildML

Perceptrons

Perceptrons

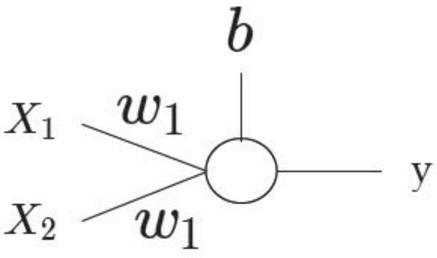
- Let’s put this all together
- Our first network will be a single neuron that will learn a simple function



X1	X2	y
0	0	0
0	1	1
1	0	1
1	1	1

Perceptrons

- How do we make a prediction for each observation?



Assume the following values:

w1	w2	b
1	-1	-0.5

Observations

X1	X2	y
0	0	0
0	1	1
1	0	1
1	1	1

Predictions

- For the first observation, $X_1 = 0, X_2 = 0, y = 0$
- First compute the weighted sum:

$$h = w_1 * X_1 + w_2 * X_2 + b$$

$$h = 1 * 0 + -1 * 0 + (-0.5)$$

$$h = -0.5$$

Assume the following values:

w1	w2	b
1	-1	-0.5

Observations

X1	X2	y
0	0	0
0	1	1
1	0	1
1	1	1

Predictions

- For the first observation, $X_1 = 0, X_2 = 0, y = 0$
- First compute the weighted sum:

$$h = w_1 * X_1 + w_2 * X_2 + b$$

$$h = 1 * 0 + -1 * 0 + (-0.5)$$

$$h = -0.5$$

→

Transform to a probability:

$$p = \frac{1}{1+\exp(-h)}$$

$$p = \frac{1}{1+\exp(-0.5)}$$

$$p = 0.38$$

Assume the following values:

w1	w2	b
1	-1	-0.5

Predictions

- For the first observation, $X_1 = 0, X_2 = 0, y = 0$
- First compute the weighted sum:

$$h = w_1 * X_1 + w_2 * X_2 + b$$

$$h = 1 * 0 + -1 * 0 + (-0.5)$$

$$h = -0.5$$

Transform to a probability:

$$p = \frac{1}{1+\exp(-h)}$$

$$p = \frac{1}{1+\exp(-0.5)}$$

$$p = 0.38$$

Assume the following values:

w1	w2	b
1	-1	-0.5

Round to get prediction:

$$\hat{y} = \text{round}(p)$$

$$\hat{y} = 0$$

Predictions

$$h = w_1 * X_1 + w_2 * X_2 + b$$

$$p = \frac{1}{1+\exp(-h)}$$

$$\hat{y} = \text{round}(p)$$

Assume the following values:

w1	w2	b
1	-1	-0.5

Complete the table:

X1	X2	y	h	p	$\hat{y}$
0	0	0	-0.5	0.38	0
0	1	1			
1	0	1			
1	1	1			

Predictions

$$h = w_1 * X_1 + w_2 * X_2 + b$$

$$p = \frac{1}{1+\exp(-h)}$$

$$\hat{y} = \text{round}(p)$$

Assume the following values:

w1	w2	b
1	-1	-0.5

Complete the table:

X1	X2	y	h	p	$\hat{y}$
0	0	0	-0.5	0.38	0
0	1	1	-1.5	0.18	0
1	0	1	0.5	0.38	0
1	1	1	-0.5	0.38	0

Performance

- Our network isn’t so great
- How do we make it better?
- What does *better* mean?
    - Need to define a measure of performance
    - There are many ways

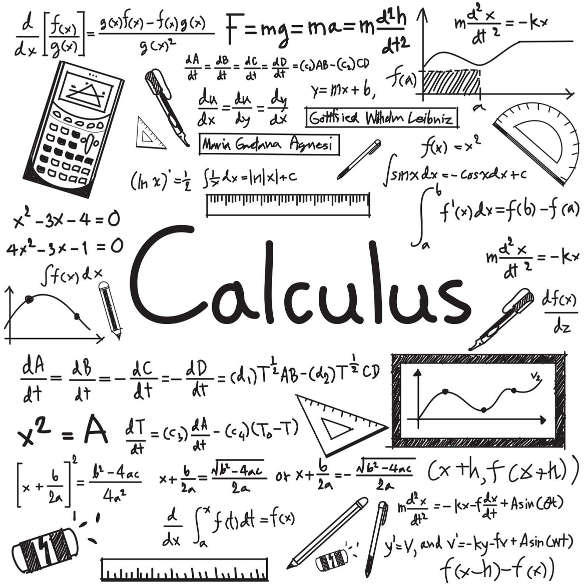
- Let’s begin with squared error: $(y - p)^2$
- We need to find values for w_1, w_2, b that make this error as small as possible.
- We need to **learn** values for w_1, w_2, b such that the difference between the predicted and actual values is as small as possible.

Learning From Data

How do we find the best values for w_1, w_2, b ?

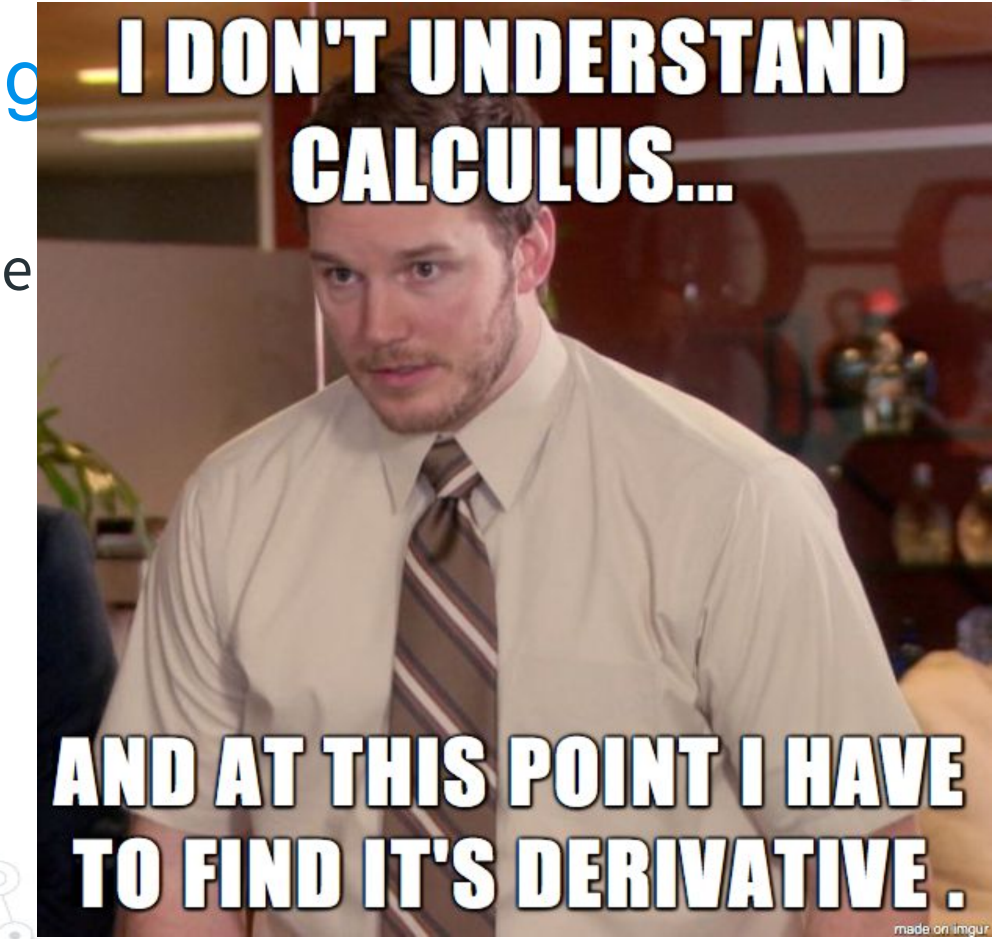
Learning From Data

How do we find the best values for w_1, w_2, b ?



Learning

How do we

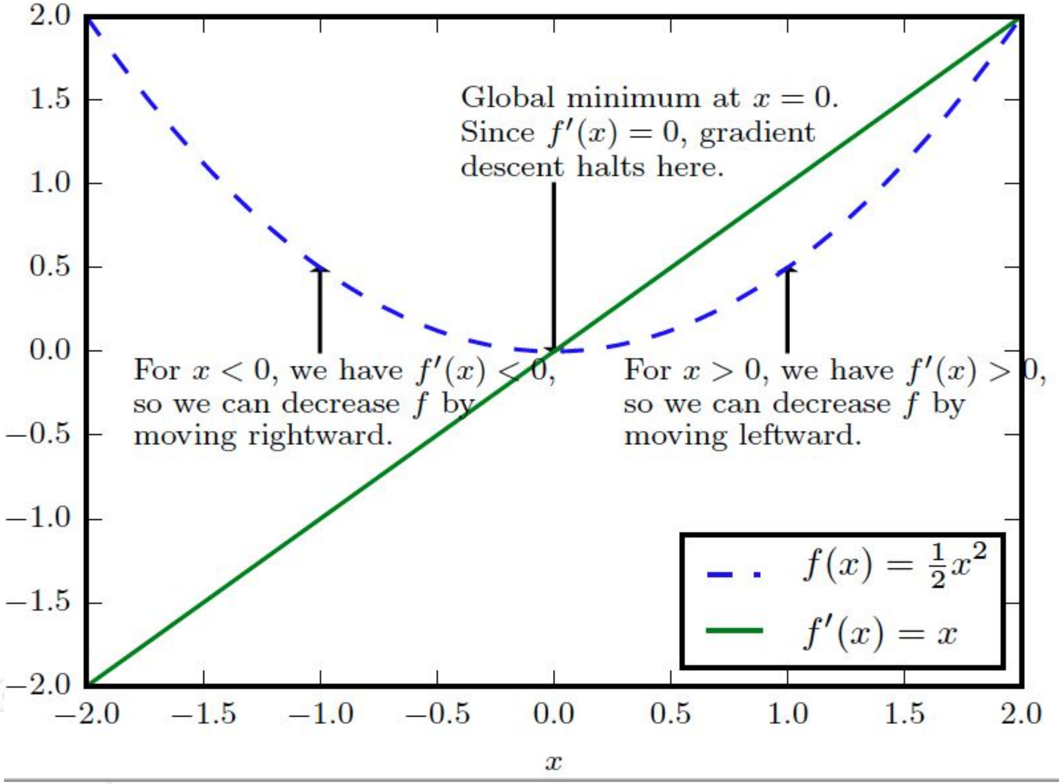

I DON'T UNDERSTAND CALCULUS...
AND AT THIS POINT I HAVE TO FIND IT'S DERIVATIVE .
made on imgur

Learning From Data

- Recall that the derivative of a function tells you how it is changing at any given location.
  - If the derivative is positive, it means it's going up
  - If the derivative is negative, it means it's going down

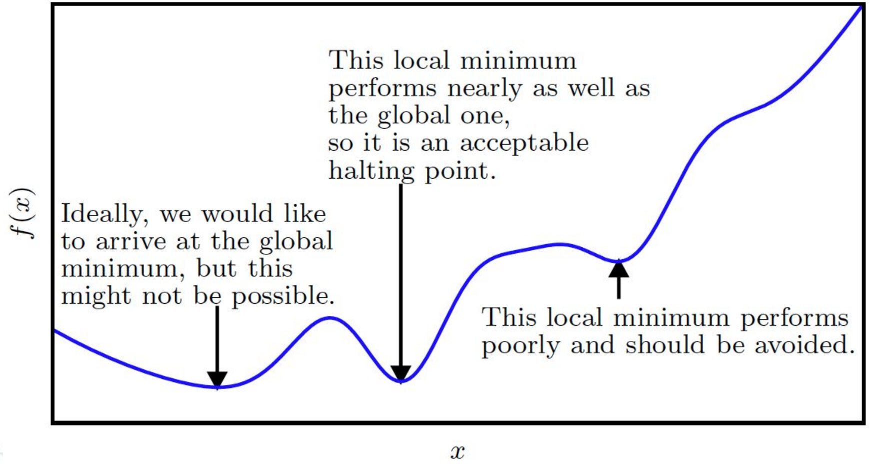
- Strategy:
  - Start with initial values for w_1, w_2, b
  - Take partial derivatives of the loss function with respect to w_1, w_2, b
  - Subtract the derivative (also called the **gradient**) from each
  - This is known as **gradient descent**

Gradient-Based Optimization



Gradient-Based Optimization

- A point that obtains the absolute lowest value of *f(x)* is a global minimum
- There may be one global minimum or multiple global minima
- It is also possible for there to be local minima that are not globally optimal
- It is common in many settings to settle for a value *f* that is very low but not necessarily minimal



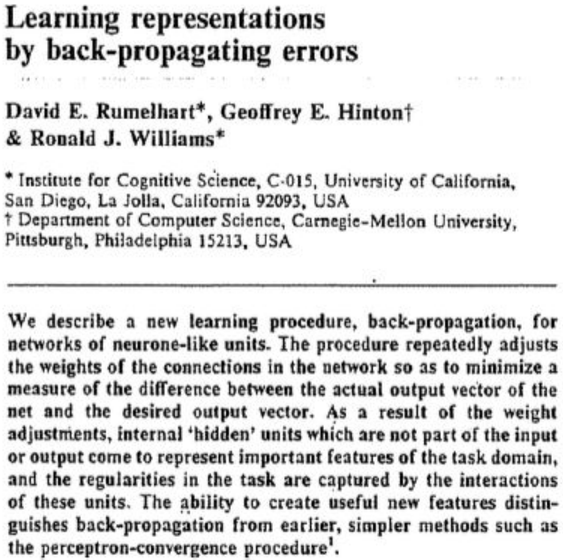
Gradient-Based Optimization

- To minimize *f*, we would like to find the direction in which *f* decreases the fastest
- It can be shown that the gradient points directly uphill and the negative gradient directly downhill
- We can therefore decrease *f* by moving in the direction of the negative gradient
- For example, for a weight w_i

$$w_i^{\text{new}} = w_i^{\text{old}} - \eta g$$

where η is the **learning rate** (how fast you want to move down the gradient), and g is the gradient

The Backpropagation Algorithm



Learning representations by back-propagating errors

David E. Rumelhart*, Geoffrey E. Hinton† & Ronald J. Williams*

* Institute for Cognitive Science, C-015, University of California,
San Diego, La Jolla, California 92093, USA
† Department of Computer Science, Carnegie-Mellon University,
Pittsburgh, Philadelphia 15213, USA

We describe a new learning procedure, back-propagation, for networks of neurone-like units. The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. As a result of the weight adjustments, internal 'hidden' units which are not part of the input or output come to represent important features of the task domain, and the regularities in the task are captured by the interactions of these units. The ability to create useful new features distinguishes back-propagation from earlier, simpler methods such as the perceptron-convergence procedure[1].

The Backpropagation Algorithm

- Our perceptron performs the following computations:

$$h = w_1 * X_1 + w_2 * X_2 + b$$

$$p = \frac{1}{1+\exp(-h)}$$

- We want to minimize this quantity:

$$l = (y - p)^2$$

- We’ll compute the gradients for each parameter by “backpropagating” errors through each component of the network

The Backpropagation Algorithm

For w_1 we need to compute

$$\frac{\partial l}{\partial w_1}$$

To get there we will use the chain rule

$$\frac{\partial l}{\partial w_1} = \frac{\partial l}{\partial p} * \frac{\partial p}{\partial h} * \frac{\partial h}{\partial w_1}$$

This is “backprop”

Computations

$$h = w_1 * X_1 + w_2 * X_2 + b$$

$$p = \frac{1}{1+\exp(-h)}$$

Loss

$$l = (y - p)^2$$

The Backpropagation Algorithm

$$\frac{\partial l}{\partial w_1} = \frac{\partial l}{\partial p} * \frac{\partial p}{\partial h} * \frac{\partial h}{\partial w_1}$$

$$\boxed{\frac{\partial l}{\partial p}} =$$

Computations

$$h = w_1 * X_1 + w_2 * X_2 + b$$

$$p = \frac{1}{1+\exp(-h)}$$

Loss

$$\boxed{l = (y - p)^2}$$

The Backpropagation Algorithm

$$\frac{\partial l}{\partial w_1} = \frac{\partial l}{\partial p} * \frac{\partial p}{\partial h} * \frac{\partial h}{\partial w_1}$$

$$\boxed{\frac{\partial l}{\partial p}} = 2 * (p - y)$$

Computations

$$h = w_1 * X_1 + w_2 * X_2 + b$$

$$p = \frac{1}{1+\exp(-h)}$$

Loss

$$\boxed{l = (y - p)^2}$$

The Backpropagation Algorithm

$$\frac{\partial l}{\partial w_1} = \frac{\partial l}{\partial p} * \frac{\partial p}{\partial h} * \frac{\partial h}{\partial w_1}$$

$$\boxed{\frac{\partial l}{\partial p}} = 2 * (p - y)$$

$$\boxed{\frac{\partial p}{\partial h}} =$$

Computations

$$h = w_1 * X_1 + w_2 * X_2 + b$$

$$\boxed{p = \frac{1}{1+\exp(-h)}}$$

Loss

$$\boxed{l = (y - p)^2}$$

The Backpropagation Algorithm

$$\frac{\partial l}{\partial w_1} = \frac{\partial l}{\partial p} * \frac{\partial p}{\partial h} * \frac{\partial h}{\partial w_1}$$

$$\boxed{\frac{\partial l}{\partial p}} = 2 * (p - y)$$

$$\boxed{\frac{\partial p}{\partial h}} = p * (1 - p)$$

Computations

$$h = w_1 * X_1 + w_2 * X_2 + b$$

$$\boxed{p = \frac{1}{1+\exp(-h)}}$$

Loss

$$\boxed{l = (y - p)^2}$$

The Backpropagation Algorithm

$$\frac{\partial l}{\partial w_1} = \frac{\partial l}{\partial p} * \frac{\partial p}{\partial h} * \frac{\partial h}{\partial w_1}$$

$$\boxed{\frac{\partial l}{\partial p}} = 2 * (p - y)$$

$$\boxed{\frac{\partial p}{\partial h}} = p * (1 - p)$$

$$\boxed{\frac{\partial h}{\partial w_1}} =$$

Computations

$$\boxed{h = w_1 * X_1 + w_2 * X_2 + b}$$

$$\boxed{p = \frac{1}{1+\exp(-h)}}$$

Loss

$$\boxed{l = (y - p)^2}$$

The Backpropagation Algorithm

$$\frac{\partial l}{\partial w_1} = \frac{\partial l}{\partial p} * \frac{\partial p}{\partial h} * \frac{\partial h}{\partial w_1}$$

$$\boxed{\frac{\partial l}{\partial p}} = 2 * (p - y)$$

$$\boxed{\frac{\partial p}{\partial h}} = p * (1 - p)$$

$$\boxed{\frac{\partial h}{\partial w_1}} = X_1$$

Computations

$$\boxed{h = w_1 * X_1 + w_2 * X_2 + b}$$

$$\boxed{p = \frac{1}{1+\exp(-h)}}$$

Loss

$$\boxed{l = (y - p)^2}$$

Putting it all together:

$$\frac{\partial l}{\partial w_1} = 2 * (p - y) * p * (1 - p) * X_1$$

Gradient Descent with Backprop

For some number of iterations:

1. Compute the gradient for w_1
2. Update $w_i^{\text{new}} = w_i^{\text{old}} - \eta g$
3. Repeat until “convergence”

Do this for each weight and bias term.

Multilayer Perceptrons

Perceptron → MLP

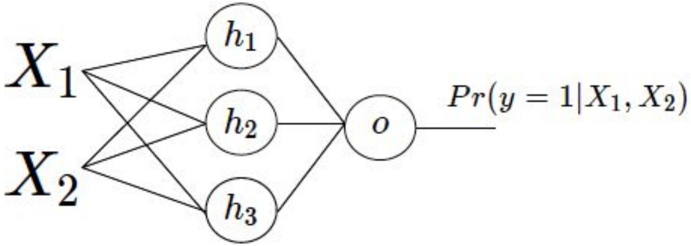
We can turn our perceptron model into a multilayer perceptron

- Instead of just one linear combination, we are going to take several, each with a different set of weights
- Each linear combination will be followed by a nonlinear activation
- Each of these nonlinear features will be fed into the logistic regression classifier (binary classifier)
- All of the weights are learned end-to-end via SGD

MLPs learn a set of nonlinear features directly from data - “feature engineering” is the hallmark of deep learning approaches

Multilayer Perceptrons (MLPs)

Suppose we have the following MLP with 1 hidden layer that has 3 hidden units:



Each neuron in the hidden layer is going to do exactly the same thing as before.

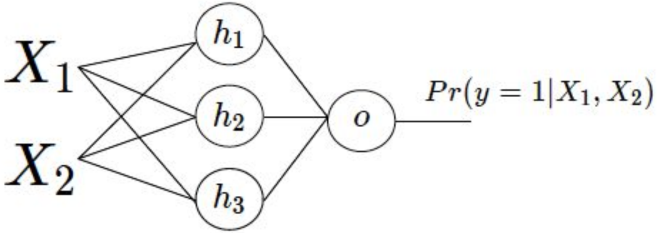
Multilayer Perceptrons (MLPs)

Computations:

$$h_j = \phi(w_{1j} * X_1 + w_{2j} * X_2 + b_j)$$

$$o = b_o + \sum_{j=1}^{3} w_{oj} * h_j$$

$$p = \frac{1}{1+\exp(-o)}$$



*If we use a sigmoid activation function

Output layer weight derivatives

$$\frac{\partial l}{\partial w_{oj}} = \frac{\partial l}{\partial p} * \frac{\partial p}{\partial o} * \frac{\partial o}{\partial w_{oj}}$$

$$= (p - y) * p * (1 - p) * h_j$$

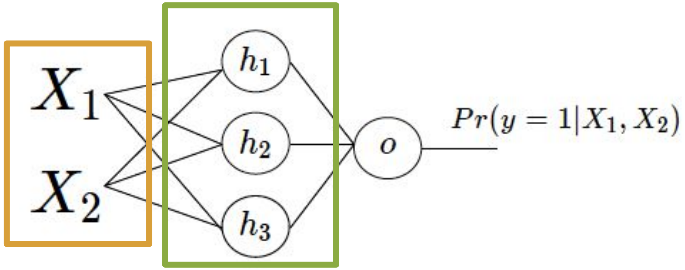
Hidden layer weight derivatives

$$\frac{\partial l}{\partial w_{1j}} = \frac{\partial l}{\partial p} * \frac{\partial p}{\partial o} * \frac{\partial o}{\partial h} * \frac{\partial h}{\partial w_{1j}}$$

$$= (p - y) * p * (1 - p) * h_j * (1 - h_j) * X_1$$

Matrix Notation

Sum notation starts to get unwieldy quickly. We can use matrix notation to represent each calculation in a more concise way.



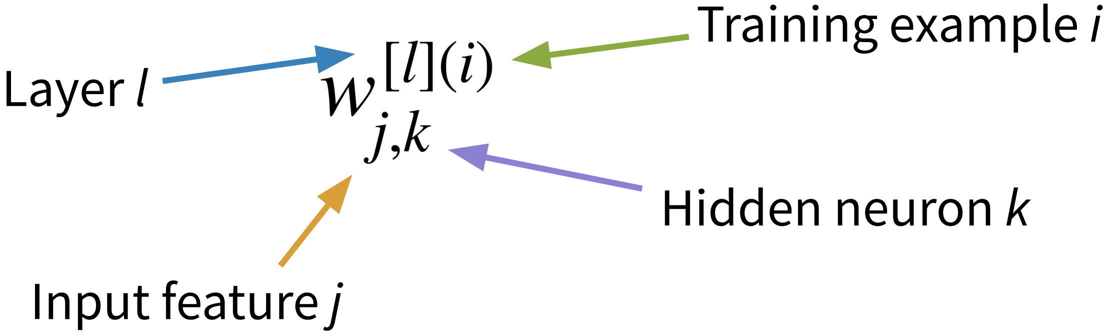
$$X = \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} \qquad W = \begin{bmatrix} w_{1,1} & w_{2,1} & w_{3,1} \\ w_{1,2} & w_{2,2} & w_{3,2} \end{bmatrix} \quad B = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

$$Z = W^T X + B \qquad H = \begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix} = \phi(Z)$$

Notation

As the number of layers grows, the number of matrices grows and we have to add a superscript to denote the layer. We also have to add a superscript to denote which training example we are referencing.

Example notation for 1 weight in 1 hidden layer for 1 training example:



$$w_{j,k}^{[l](i)}$$

Layer *l*

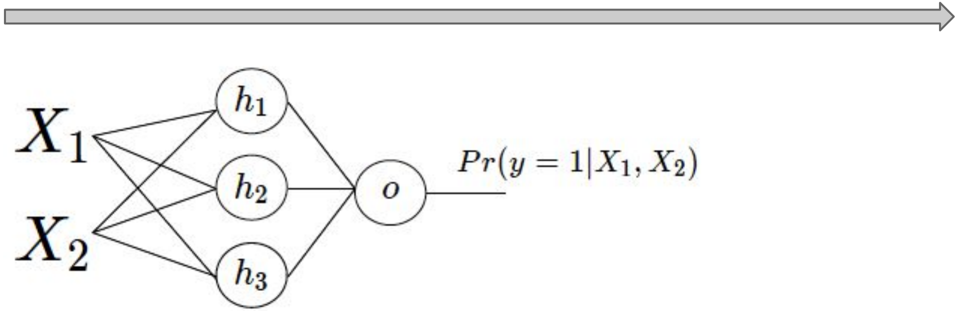
Training example *i*

Hidden neuron *k*

Input feature *j*

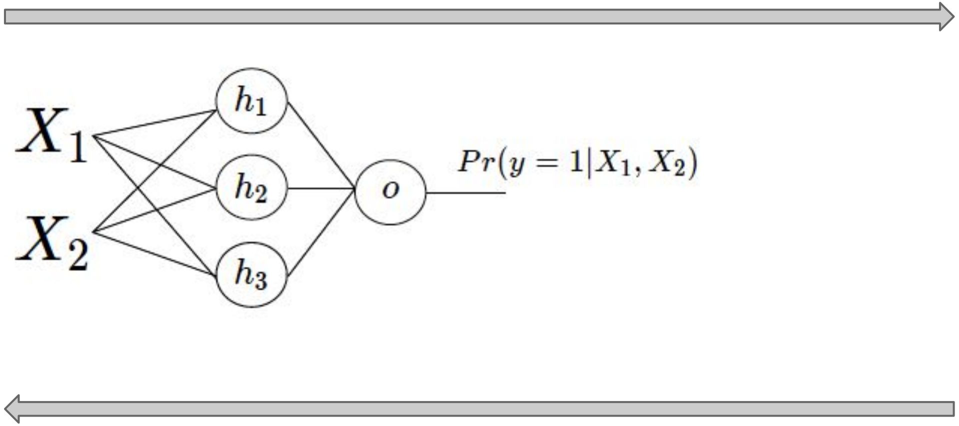
MLP Terminology

Forward pass = computing probability from input



MLP Terminology

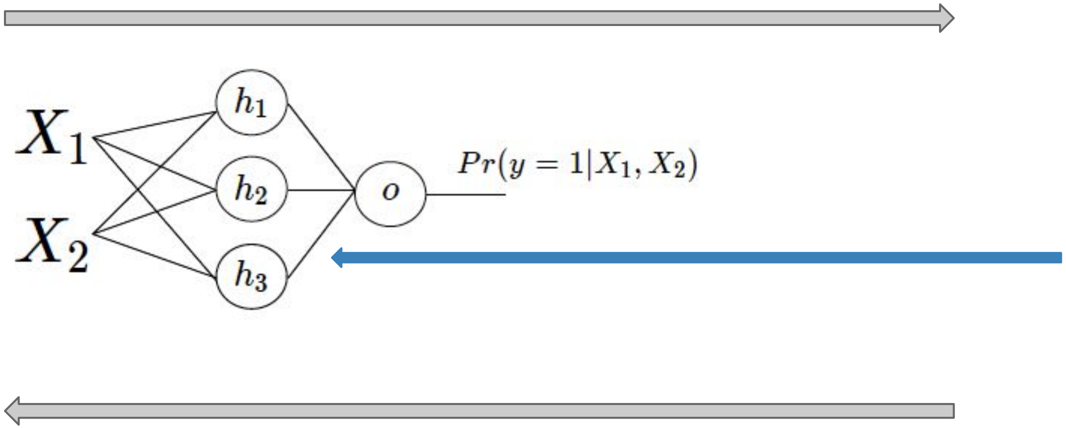
Forward pass = computing probability from input



Backward pass = computing derivatives from the output

MLP Terminology

Forward pass = computing probability from input

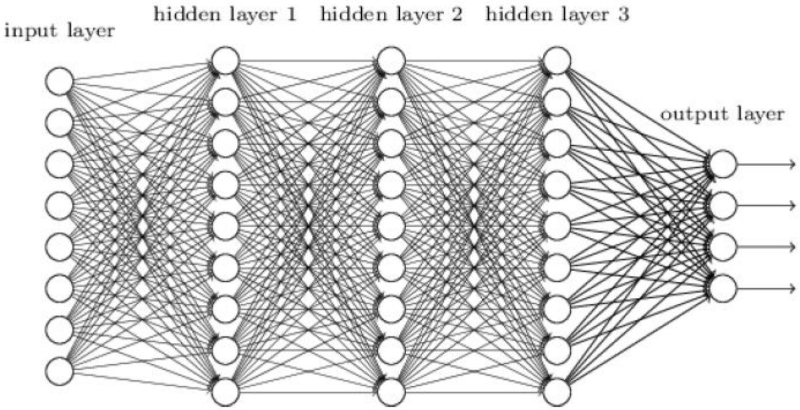


Hidden layers are also called “dense” layers or “fully connected” layers

Backward pass = computing derivatives from the output

MLPs

Increasing the number of layers increases the flexibility of the model - but run the risk of overfitting



Conclusions

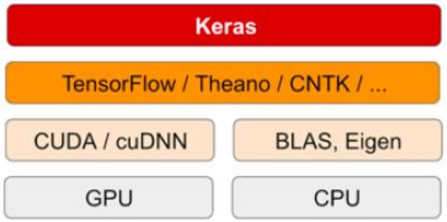
- Backprop, perceptrons, and MLPs are the building blocks of neural nets

- You’ll get a chance to demonstrate your mastery in Problem Set 1

- We will use these concepts for the rest of the semester

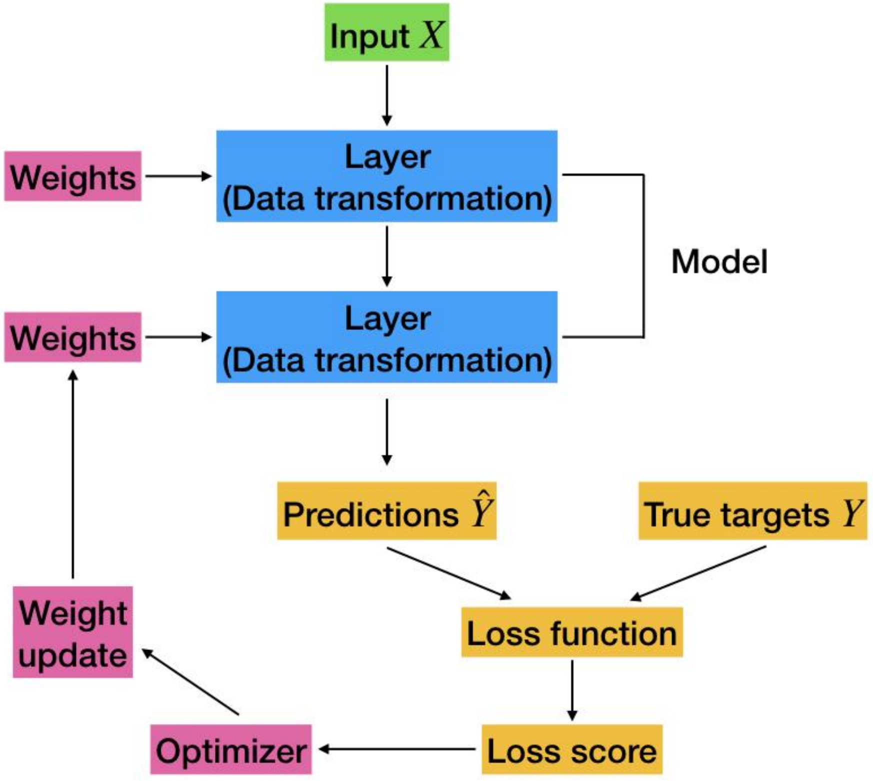
Coding
Neural Nets

Keras and Tensorflow



- Keras is a model-level library that provides high-level building blocks for developing deep learning models
- It doesn’t handle low-level operations like matrix and tensor (n-dimensional matrix) multiplication and differentiation
  - It uses TensorFlow or Theano or CNTK (Microsoft Cognitive Toolkit) backends for this
  - We will be using TensorFlow
    - It is the most widely adopted, scalable and production ready
- Keras can run on both CPUs and GPUs
  - When running on CPUs, uses Eigen for tensor operations
  - When running on GPUs, uses the NVIDIA CUDA Deep Neural Network library (cuDNN)

Neural Network Workflow

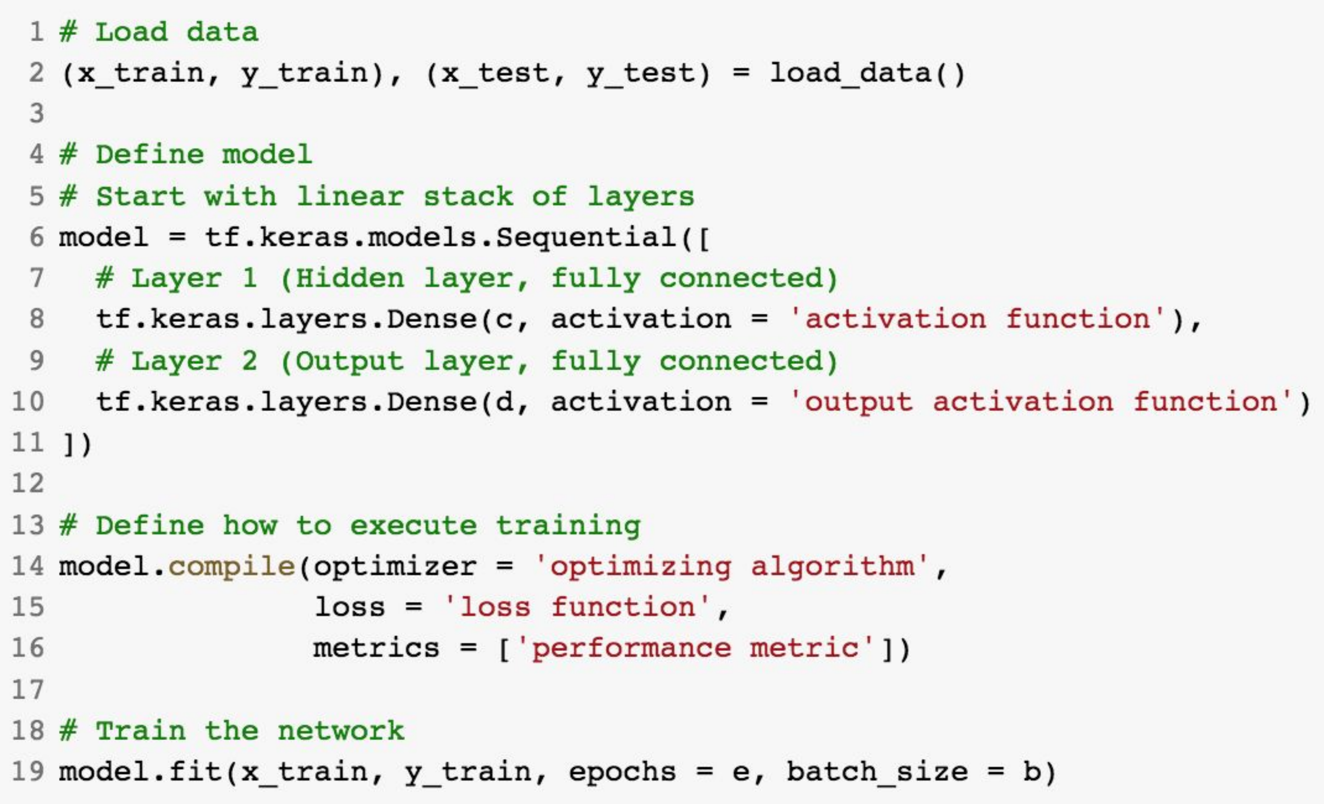


Generic Feedforward Network

Elements needed:

1. Necessary libraries
2. Dataset split into training and test sets (validation as well if you have enough data)
3. **`models.sequential():`** defines a linear, or sequential architecture made up of a set of layers that will stack to create the network
4. **`layers.Dense():`** specifies a fully connected layer
5. **`model.compile(optimizer, loss, metrics):`** specifies how to execute the training of the network
6. **`model.fit(train_data, train_labels, epochs, batch_size):`** fits the neural net using the training data, runs for a specified number of iterations (epochs) using batch_size number of training examples at a time

Generic Feedforward Network

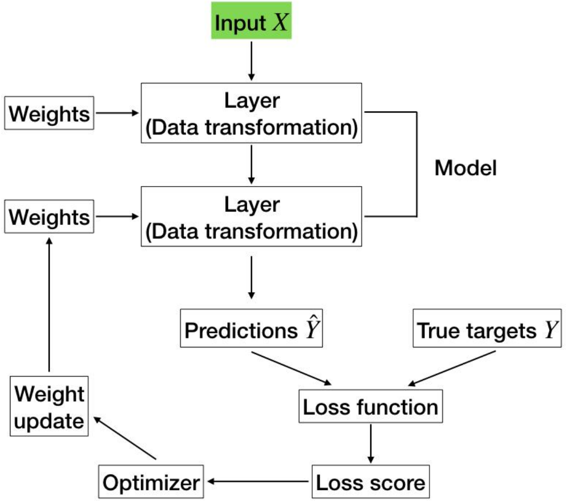


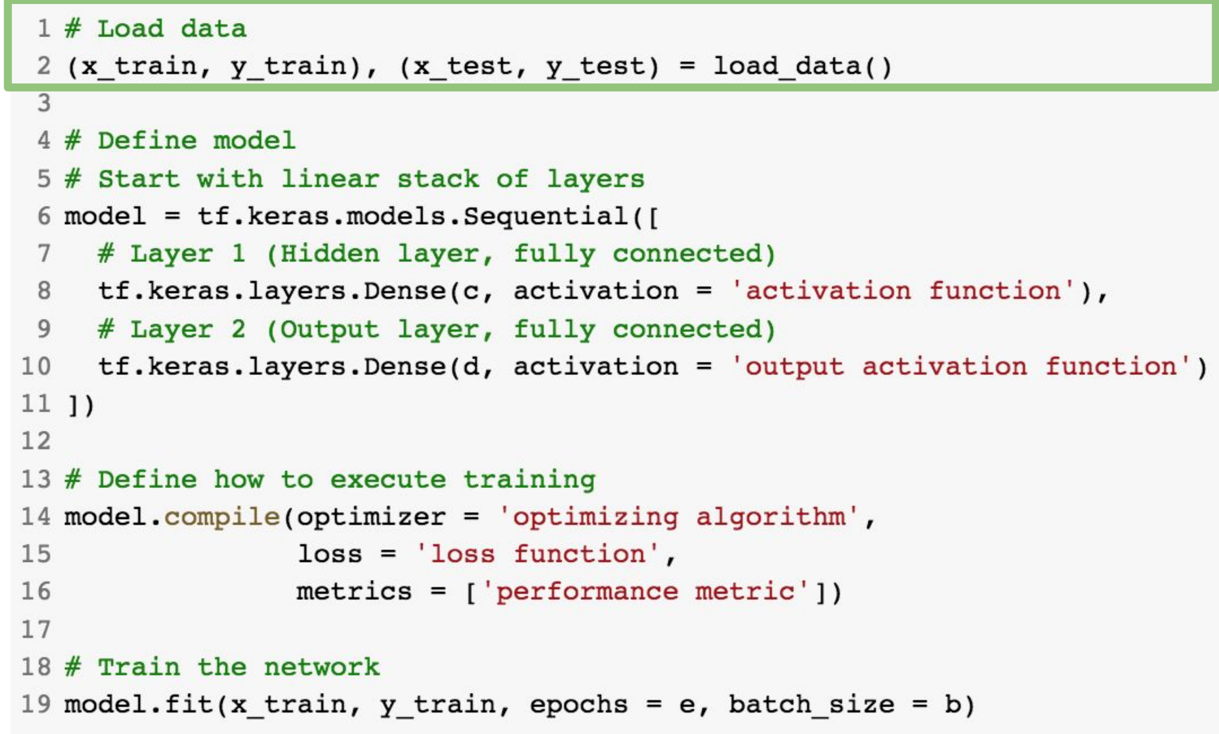
```python
# Load data
(x_train, y_train), (x_test, y_test) = load_data()

# Define model
# Start with linear stack of layers
model = tf.keras.models.Sequential([
  # Layer 1 (Hidden layer, fully connected)
  tf.keras.layers.Dense(c, activation = 'activation function'),
  # Layer 2 (Output layer, fully connected)
  tf.keras.layers.Dense(d, activation = 'output activation function')
])

# Define how to execute training
model.compile(optimizer = 'optimizing algorithm',
              loss = 'loss function',
              metrics = ['performance metric'])

# Train the network
model.fit(x_train, y_train, epochs = e, batch_size = b)
```



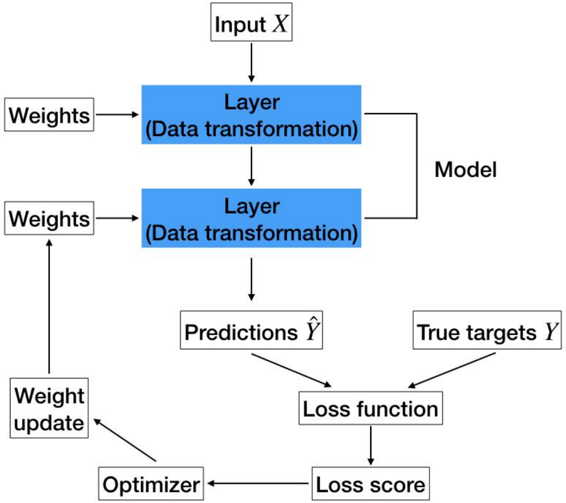


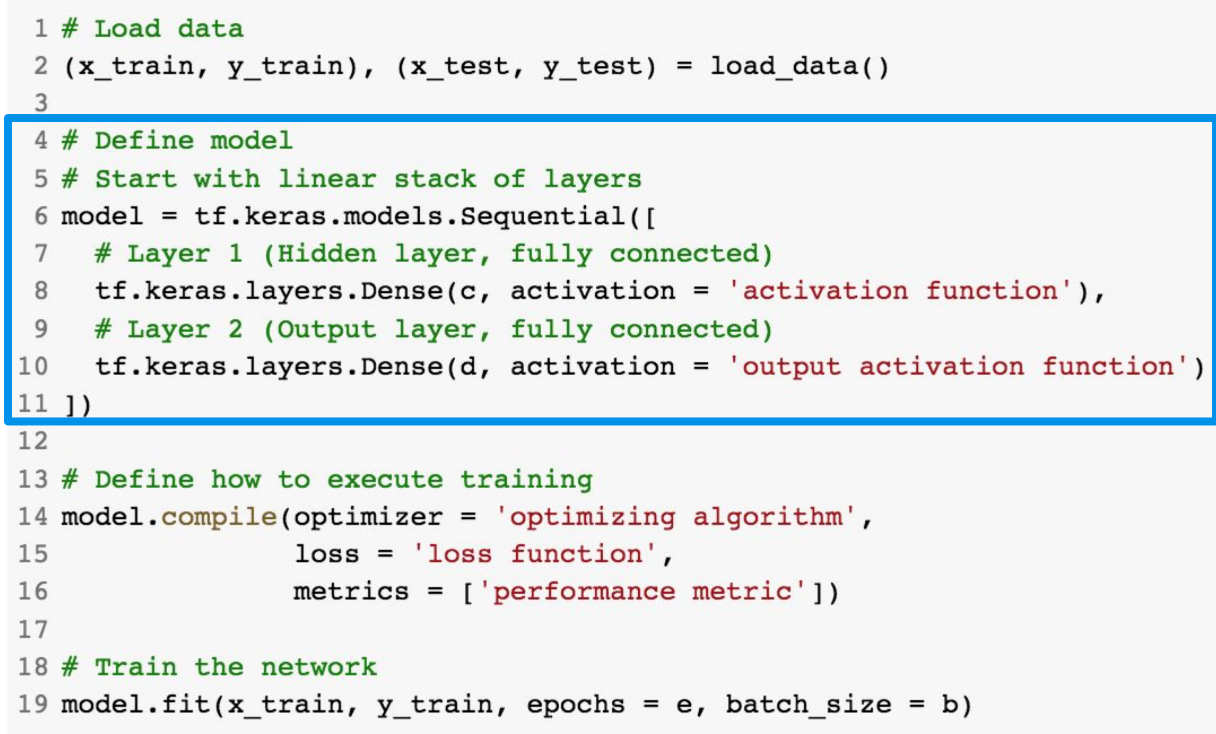
```
# Load data
(x_train, y_train), (x_test, y_test) = load_data()

# Define model
# Start with linear stack of layers
model = tf.keras.models.Sequential([
  # Layer 1 (Hidden layer, fully connected)
  tf.keras.layers.Dense(c, activation = 'activation function'),
  # Layer 2 (Output layer, fully connected)
  tf.keras.layers.Dense(d, activation = 'output activation function')
])

# Define how to execute training
model.compile(optimizer = 'optimizing algorithm',
              loss = 'loss function',
              metrics = ['performance metric'])

# Train the network
model.fit(x_train, y_train, epochs = e, batch_size = b)
```



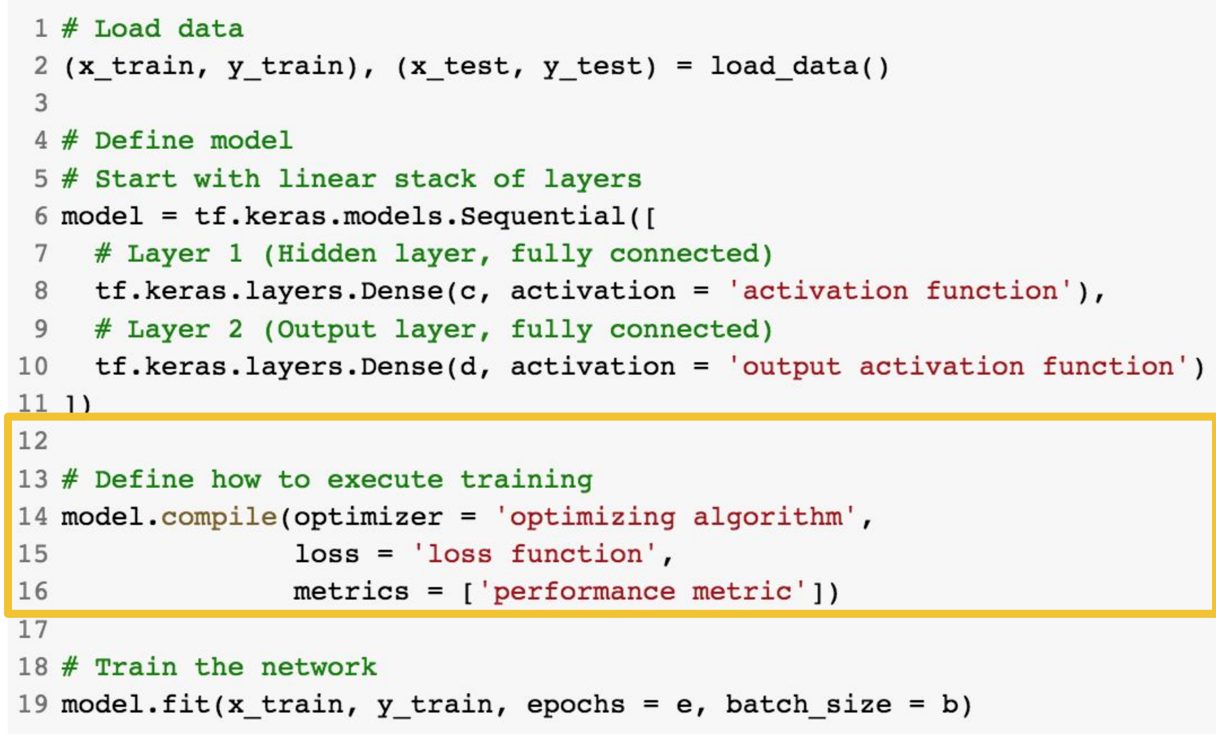


```
# Load data
(x_train, y_train), (x_test, y_test) = load_data()

# Define model
# Start with linear stack of layers
model = tf.keras.models.Sequential([
  # Layer 1 (Hidden layer, fully connected)
  tf.keras.layers.Dense(c, activation = 'activation function'),
  # Layer 2 (Output layer, fully connected)
  tf.keras.layers.Dense(d, activation = 'output activation function')
])

# Define how to execute training
model.compile(optimizer = 'optimizing algorithm',
              loss = 'loss function',
              metrics = ['performance metric'])

# Train the network
model.fit(x_train, y_train, epochs = e, batch_size = b)
```

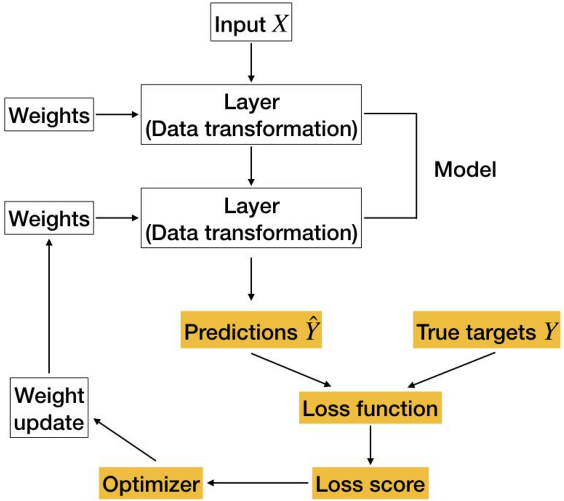


```python
# Load data
(x_train, y_train), (x_test, y_test) = load_data()

# Define model
# Start with linear stack of layers
model = tf.keras.models.Sequential([
  # Layer 1 (Hidden layer, fully connected)
  tf.keras.layers.Dense(c, activation = 'activation function'),
  # Layer 2 (Output layer, fully connected)
  tf.keras.layers.Dense(d, activation = 'output activation function')
])

# Define how to execute training
model.compile(optimizer = 'optimizing algorithm',
              loss = 'loss function',
              metrics = ['performance metric'])

# Train the network
model.fit(x_train, y_train, epochs = e, batch_size = b)
```



Input X

Weights

Layer (Data transformation)

Weights

Layer (Data transformation)

Model

Predictions $\hat{Y}$

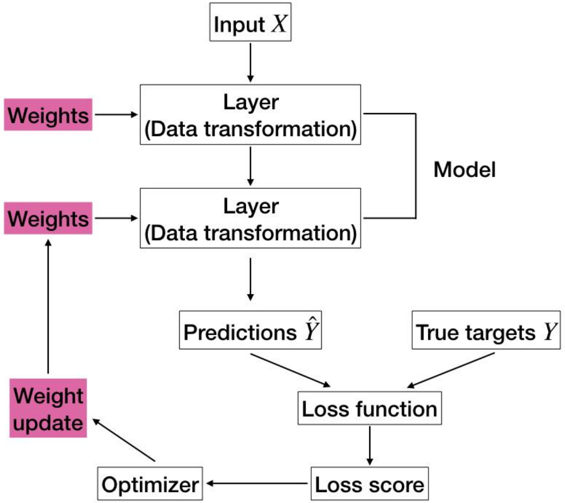
True targets Y

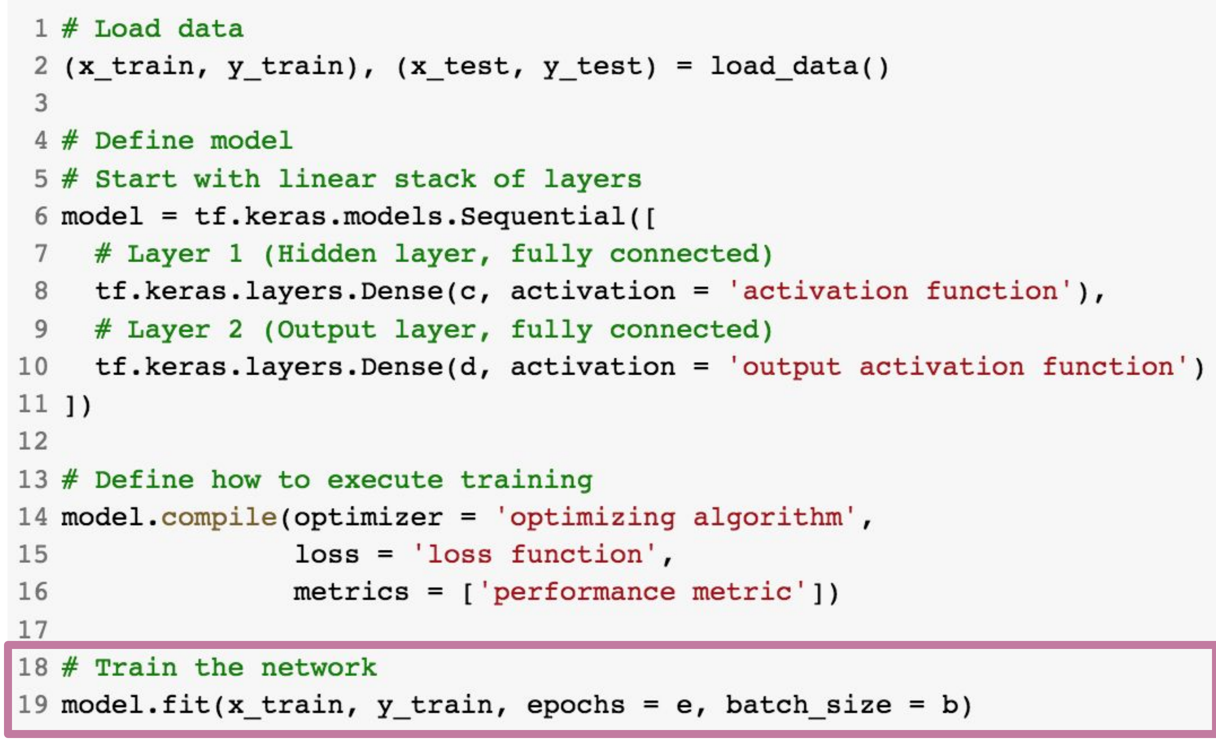
Loss function

Weight update

Loss score

Optimizer





```
# Load data
(x_train, y_train), (x_test, y_test) = load_data()

# Define model
# Start with linear stack of layers
model = tf.keras.models.Sequential([
  # Layer 1 (Hidden layer, fully connected)
  tf.keras.layers.Dense(c, activation = 'activation function'),
  # Layer 2 (Output layer, fully connected)
  tf.keras.layers.Dense(d, activation = 'output activation function')
])

# Define how to execute training
model.compile(optimizer = 'optimizing algorithm',
              loss = 'loss function',
              metrics = ['performance metric'])

# Train the network
model.fit(x_train, y_train, epochs = e, batch_size = b)
```

Generic Feedforward Network

train_data: training examples (matrix of feature vectors; $\mathbf{X}_{\mathbf{train}}$)

train_labels: training labels ($\mathbf{y}_{\mathbf{train}}$)

test_data: test examples used to measure performance of network ($\mathbf{X}_{\mathbf{test}}$)

test_labels: test set labels ($\mathbf{y}_{\mathbf{test}}$)

Optimizing algorithms: rmsprop, sgd, adagrad, adam, etc.

Loss function options: mse, mae, categorical_crossentropy, etc.

Performance measure options: accuracy, mae, etc.

Here:

$\boldsymbol{c}$ = the number of hidden units (neurons) in a hidden layer

$\boldsymbol{d}$ = the number of units (neurons) in the output layer

$\boldsymbol{e}$ = the number of epochs (iterations) over entire training data set

$\boldsymbol{b}$ = the batch size (how many training examples to optimize at once)