Fundamentals of deep learning

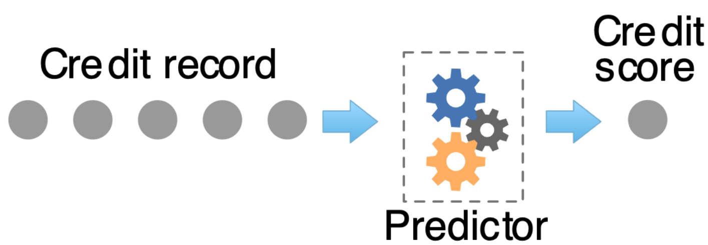
Crash course in using PyTorch to train models

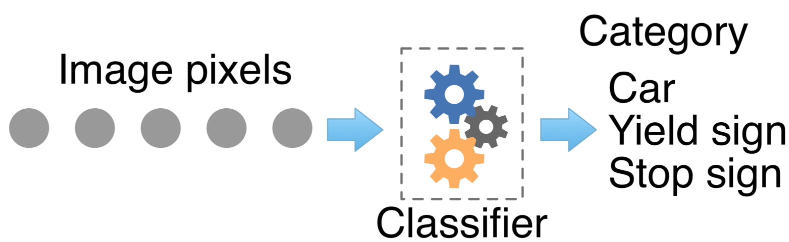
Terence Parr
MSDS program
University of San Francisco

UNIVERSITY OF SAN FRANCISCO

The goal of (supervised) machine learning

- Obtain a model that predicts numeric values (regressor) or categories (classifier) from information about an entity:





The abstraction

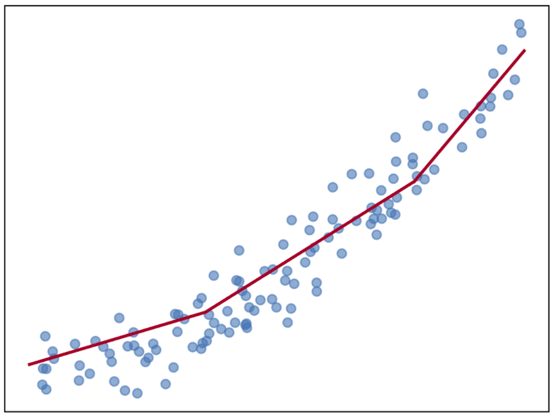
- Training a model means capturing the relationship between feature vectors and a target variable, given a training data set
- **Feature vector** $\boldsymbol{x}$: set of features or attributes characterizing an entity, such as square footage, num of bedrooms, bathrooms
- **Target** y: either a scalar value like rent price (regressor), or an integer indicating "creditworthy" or "it's not cancer" (classifier)
- Model captures this relationship: $X \rightarrow \boldsymbol{y}$

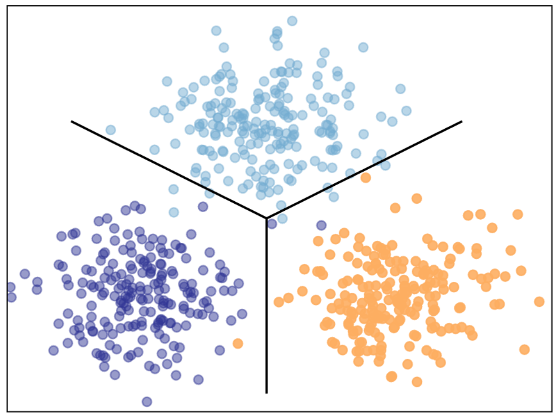
$$X = \boxed{\begin{array}{c} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_n \end{array}} \qquad \boldsymbol{y} = \boxed{\begin{array}{c} y_1 \\ y_2 \\ \vdots \\ y_n \end{array}}$$

Row vectors, $\boldsymbol{x}_i$, represent instances

Regression versus classification

- Predictors fit curves to data and classifiers draw decision boundaries between data points in different categories
- Two sides of the same coin in implementation typically





Models

- Models are composed of *parameters*; predictions are a computation based upon these parameters
- Models have *architecture; e*.g., number of layers, number of neurons per layer, which nonlinearity to use, etc…
- Models have *hyper-parameters* that govern architecture and the training process; e.g., learning rate, weight decay, drop out rate
- Hyper-parameters are specified by the programmer, not computed from the training data; must be tuned
- Deep learning training greatly affected by learning rate, and even things like parameter initialization

Training

- Training a model means finding optimal (or good enough) model parameters as measured by a *loss* (cost) function
- Loss function measures the difference between model predictions and known targets
- *Underfitting*: model unable to capture the relationship $X \rightarrow \boldsymbol{y}$ (assuming there is a relationship to be had)
- *Overfitting*: model is too specific to the training data and doesn't generalize well (e.g., home price predictor trained on just condos)
- To *generalize* means we get accurate predictions for test feature vectors not found in the training set

Terminology: Loss function vs metric

- *Loss function*: these are minimized to train a model
  E.g., gradient descent uses loss to train regularized linear model
- *Metric*: evaluate accuracy of predictions compared to known results (the business perspective)
- Both are functions of y and $\hat{y}$, but loss is also possibly model parameters (e.g., linear model regularization loss tests parameters)
- Examples:
  - Train: MSE loss & Metric: MSE metric
  - Train: MSE loss & Metric : MAE metric
  - Train: Negative Log Loss & Metric : misclassification or FP/FN metric
- If metric is applied to validation or test set, informs on generality and quality of your model

See also stackoverflow post by Chstiros Tsatsoulis: https://goo.gl/T5AmrT

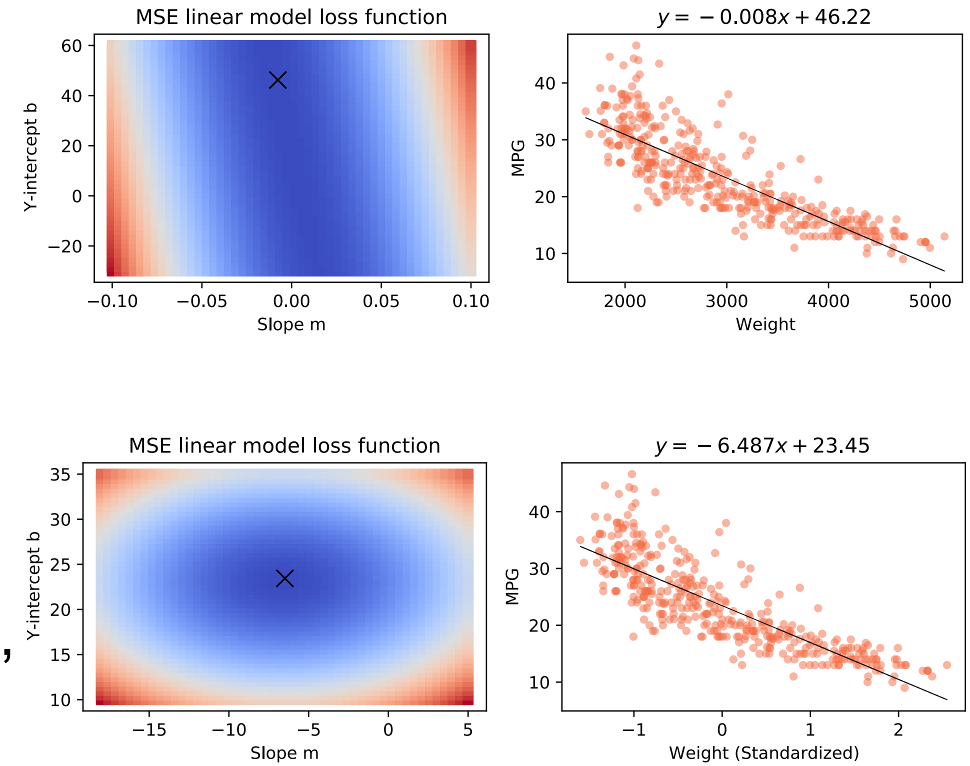
Train, validate, test

- We always need 3 data sets with known answers:
  - training
  - validation (as shorthand you'll hear me & others call this test set)
  - testing (put in a vault and **don't peek**!!)
- Validation set: used to evaluate, tune models and features
  - Any changes you make to model tailor it to this specific validation set
- Test set: used exactly **once** after you think you have best model
  - The only true measure of model's generality, how it'll perform in production
  - Never use test set to tune model
- Production: recombine all sets back into a big training set again, retrain model but don't change it according to test set metrics

Preparing data

$$X = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_n \end{bmatrix} \qquad \boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

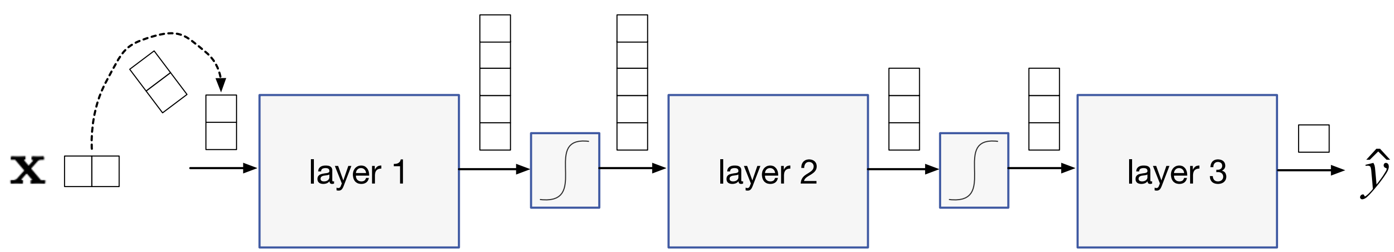
- Everything must be numeric
- No missing values
- Dummy encode categoricals
- Should normalize numeric features in X to zero-mean, variance one ("whitening")
- Speeds up training
- Compare regression equations, loss function surfaces



Deep learning regressors

What's a neural network?

- A combination of linear and nonlinear transformations
  - Linear: $z^{[layer]} = W^{[layer]}\boldsymbol{x}^{\boldsymbol{T}} + \boldsymbol{b}^{[layer]}$
  - Nonlinear example: $\boldsymbol{a}^{[layer]} = \boldsymbol{\sigma}\left(z^{[layer]}\right)$; called *activation function*
- Ignore the neural network metaphor, but know the terminology
- Networks have multiple *layers*; layer is a stack of *neurons*



- Transform raw $\boldsymbol{x}$ vector into better and better features, final linear layer can then make excellent prediction

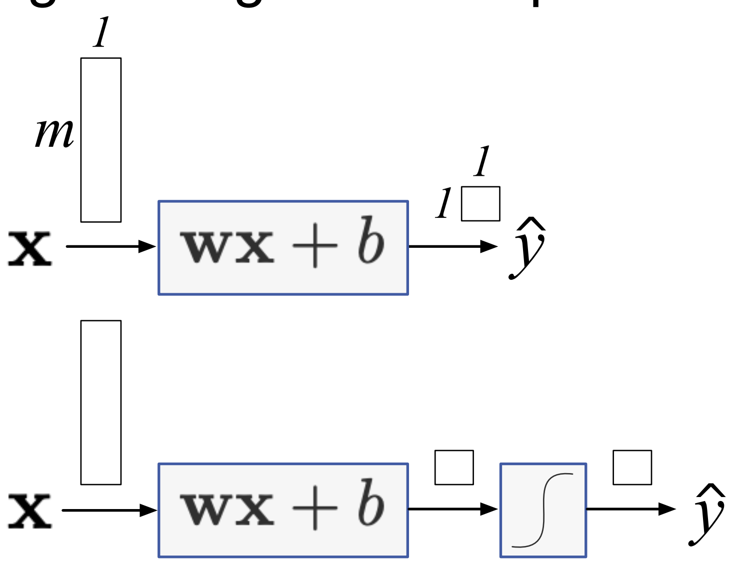
DL Building blocks

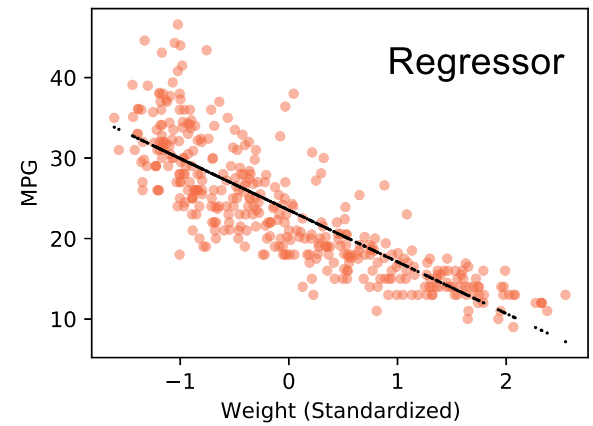
linear | sigmoid | ReLU (rectified linear unit)

- $\hat{y} = w_1x_1 + w_2x_2 + \ ... + w_mx_m + b = \boldsymbol{wx^T} + b$ for n x m dim X
- Linear/logistic regression equivalents (one $\boldsymbol{x}$ instance)

Regressor

Two-class Classifier





Underfitting a bit here
(need more of a quadratic)

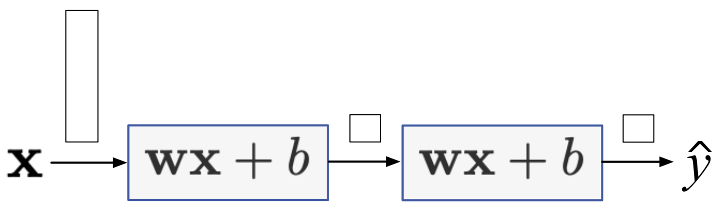
Assume we magically know $\boldsymbol{w}$ and b

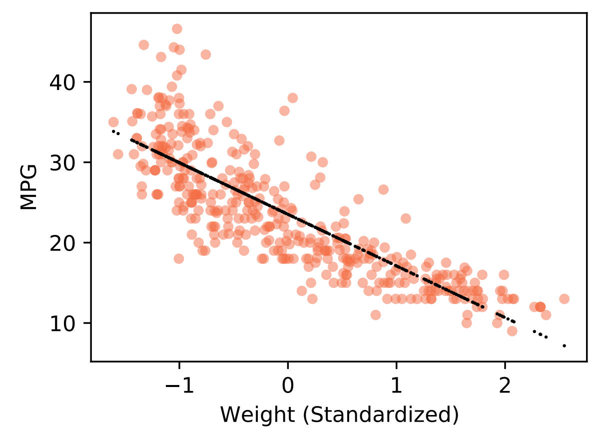
Try adding layers to get more power

- But, sequence of linear models is just a linear model

$$\hat{y} = w'(\mathbf{w}\mathbf{x}^T + b) + b' = w'\mathbf{w}\mathbf{x}^T + w'b + b' = \mathbf{w}''\mathbf{x}^T + b''$$

(w' is scalar since $\boldsymbol{wx^T} + b$ is scalar)





Still just a line

- PyTorch code

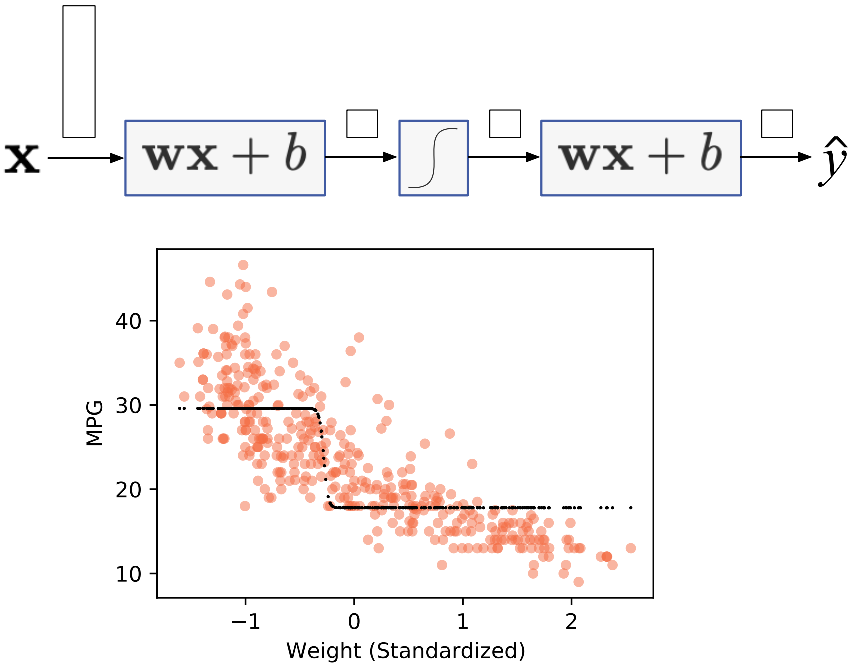
```
model = nn.Sequential(
    nn.Linear(m, 1), # m features
    nn.Linear(1, 1)
)
```

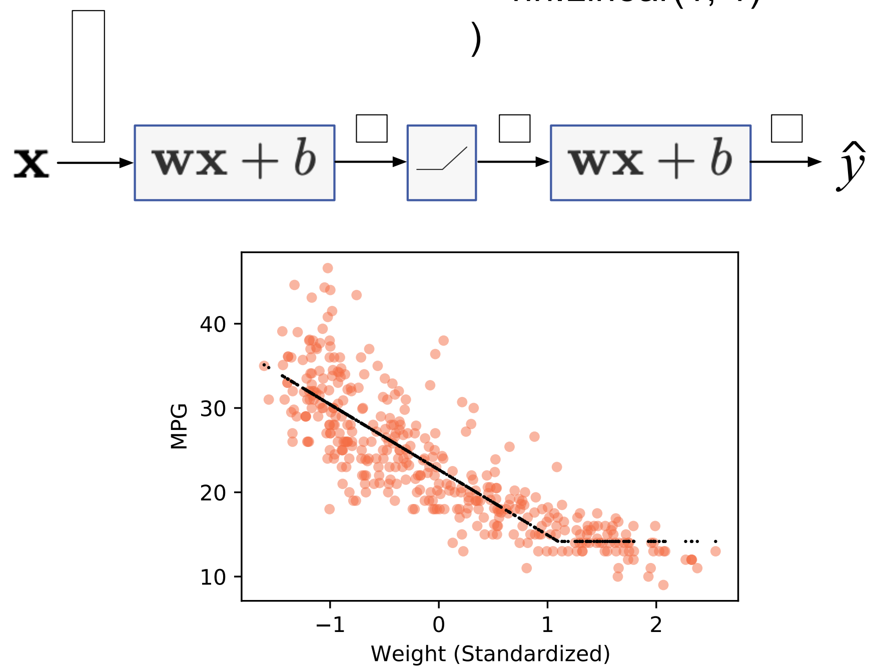
(For simplicity, I'm using proper $\boldsymbol{wx^T}$ in math but omitting transpose in diagrams)

UNIVERSITY OF SAN FRANCISCO

Must introduce nonlinearity

```
model = nn.Sequential(
    nn.Linear(m, 1),
    nn.ReLU(),
    nn.Linear(1, 1)
)
```

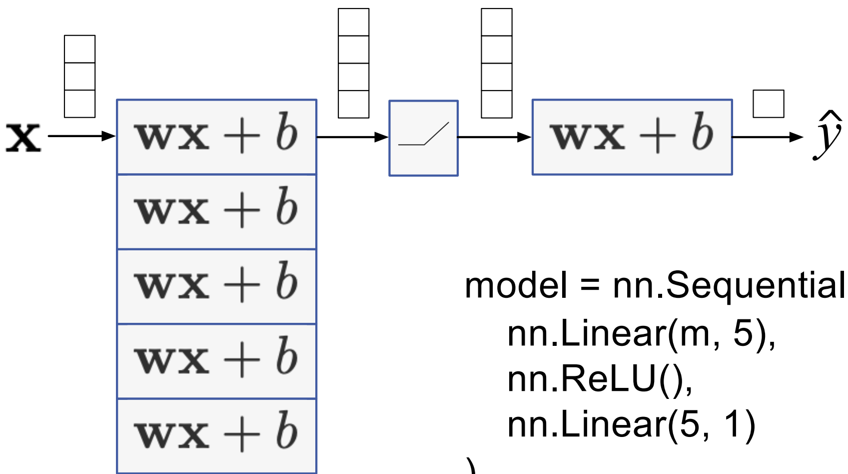


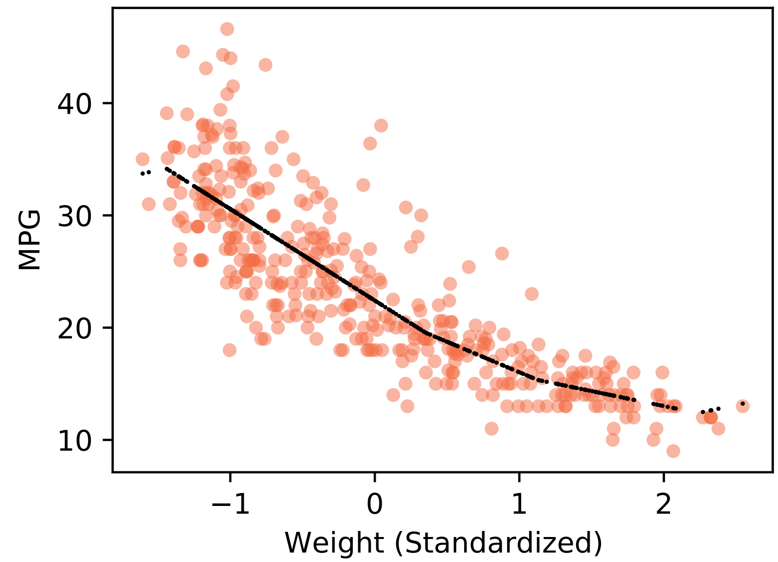


ReLU idea here: Draw two lines then clip at intersection

Stack linear models (neurons) for more power

- Stack gives layer: W matrix and $\boldsymbol{b}$
- $\boldsymbol{a}^{[1]} = relu(W^{[1]}\boldsymbol{x}^{\boldsymbol{T}} + \boldsymbol{b}^{[1]})$
- $\hat{y} = \boldsymbol{a}^{[2]} = W^{[2]}\boldsymbol{a}^{[1]} + \boldsymbol{b}^{[2]}$



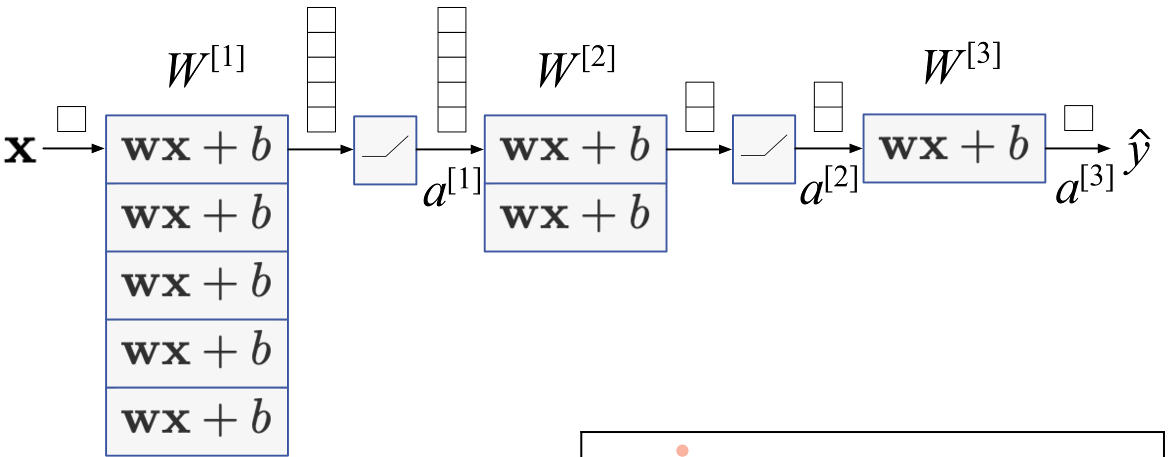


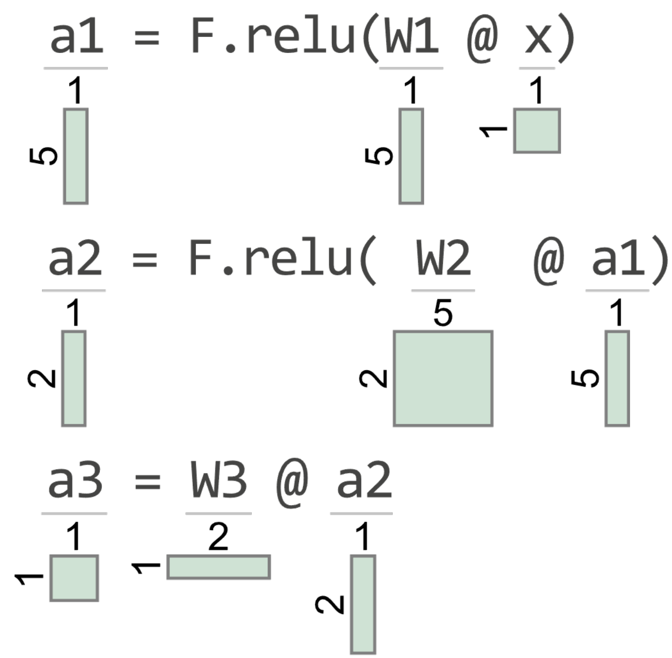

```
model = nn.Sequential(
    nn.Linear(m, 5),
    nn.ReLU(),
    nn.Linear(5, 1)
)
```

UNIVERSITY OF SAN FRANCISCO

All those $\boldsymbol{w}$ and b are different

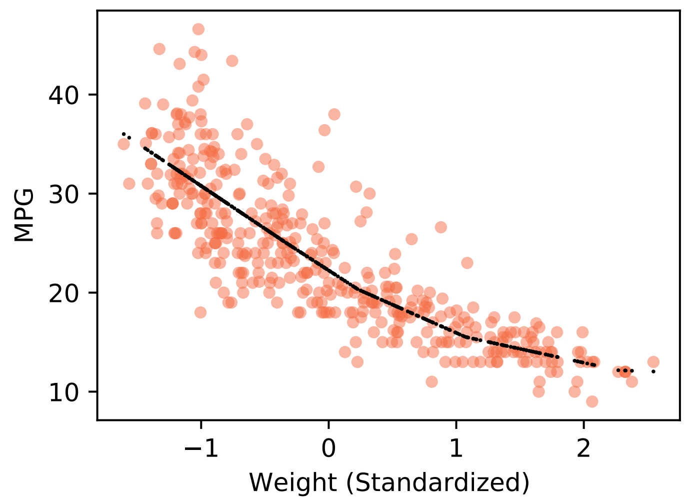
Math for Cars dataset 1D: weight→MPG





(courtesy of TensorSensor)

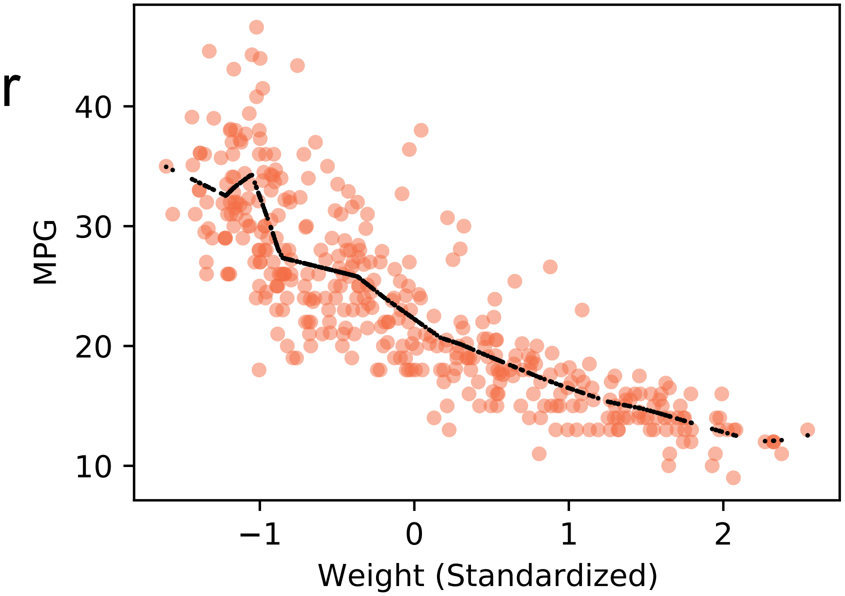
```
model = nn.Sequential(
    nn.Linear(1, 5),
    nn.ReLU(),
    nn.Linear(5, 2),
    nn.ReLU(),
    nn.Linear(2, 1)
)
```



Too much strength can lead to overfitting

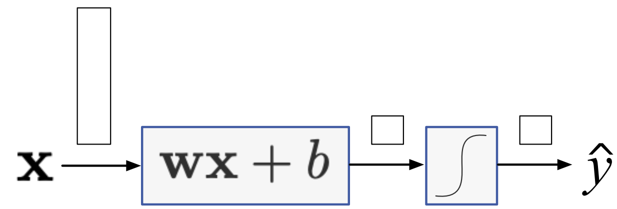
- Models with too many parameters will overfit easily, if we train a long time
- We'll look at regularization later

```
model = nn.Sequential(
    nn.Linear(1, 1000),
    nn.ReLU(),
    nn.Linear(1000, 1)
)
```

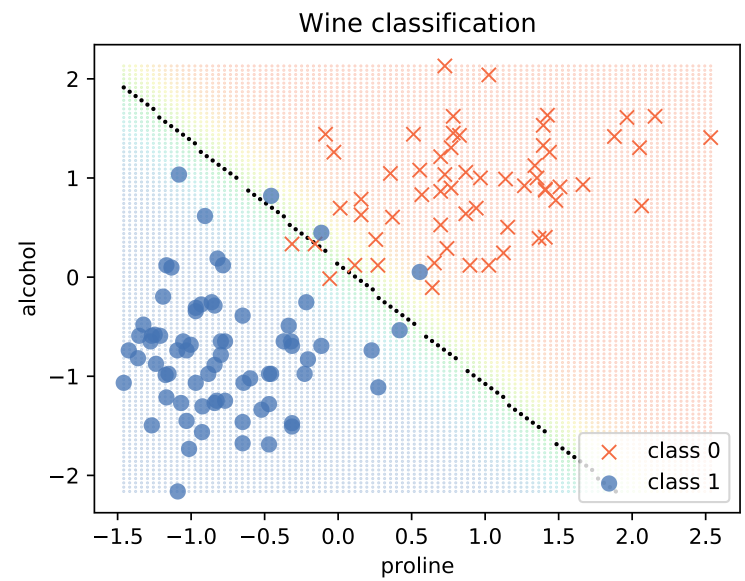


Classifiers

Binary classifiers



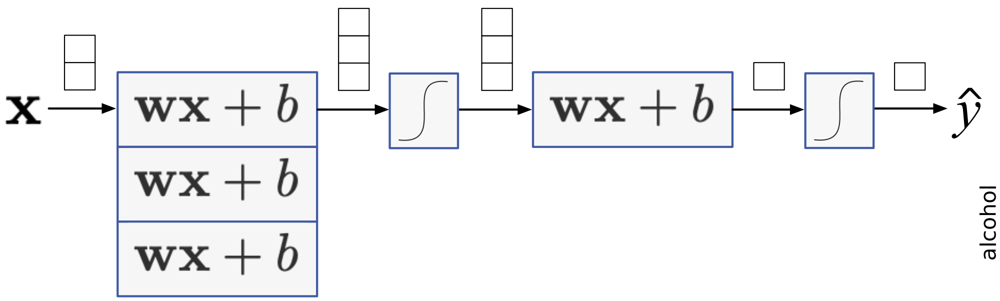
- Add sigmoid to regressor and we get a two-class classifier
- Prediction $\hat{y}$ is probability of class 1
- One-layer network with sigmoid activation function is just a logistic regression model
- Provides hyper-plane decision surfaces

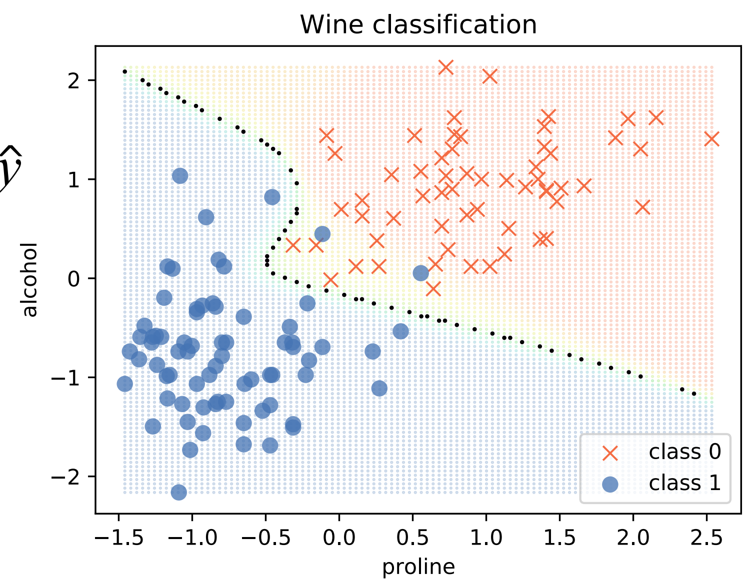



```
# 2 input vars: proline, alcohol
model = nn.Sequential(
    nn.Linear(2, 1),
    nn.Sigmoid(),
)
```

Stack neurons, add layer

- We get a nonlinear decision surface



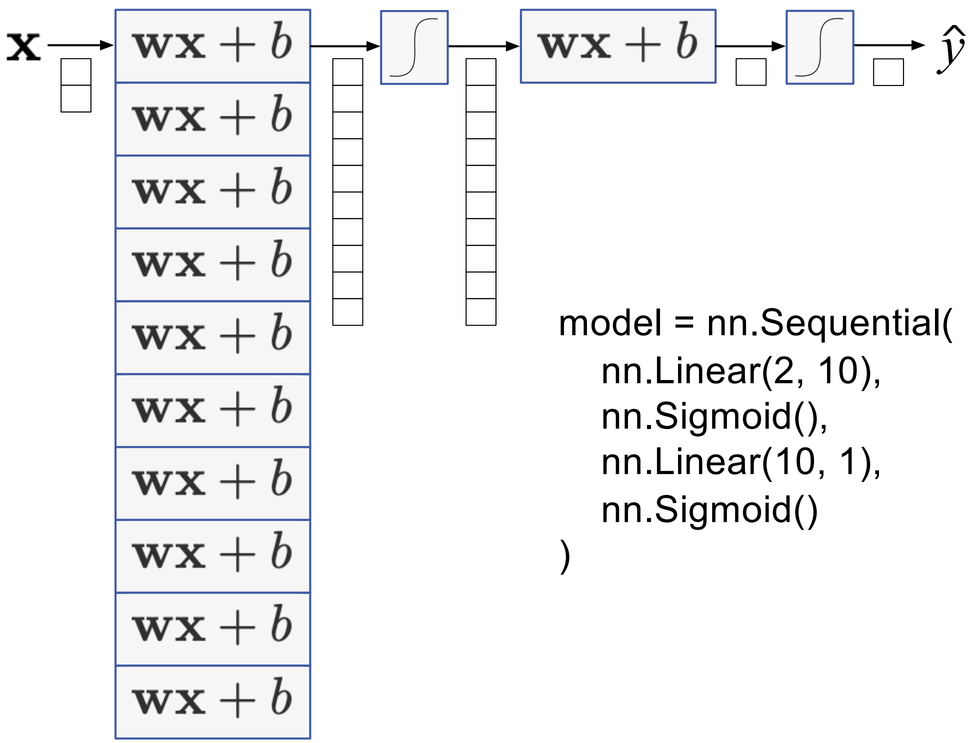



```
model = nn.Sequential(
    nn.Linear(2, 3),
    nn.Sigmoid(),
    nn.Linear(3, 1),
    nn.Sigmoid()
)
```

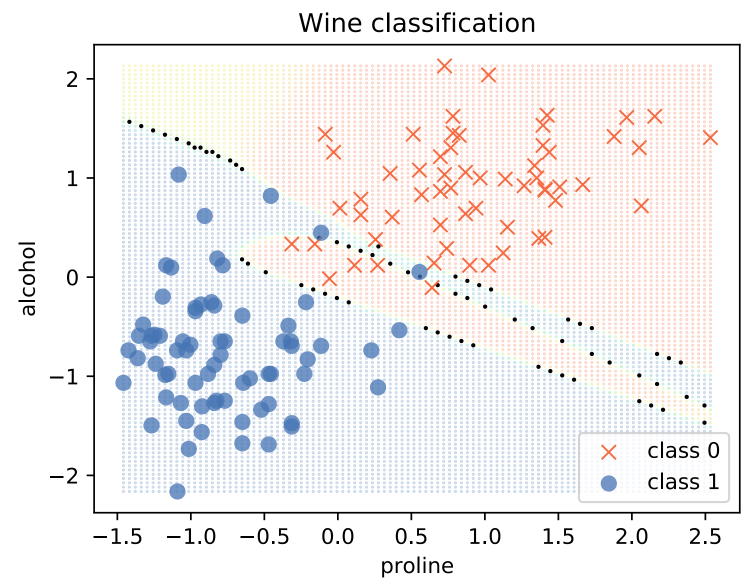
All those $\boldsymbol{w}$ and b are different

UNIVERSITY OF SAN FRANCISCO

More neurons: more complex decision surface




```
model = nn.Sequential(
    nn.Linear(2, 10),
    nn.Sigmoid(),
    nn.Linear(10, 1),
    nn.Sigmoid()
)
```

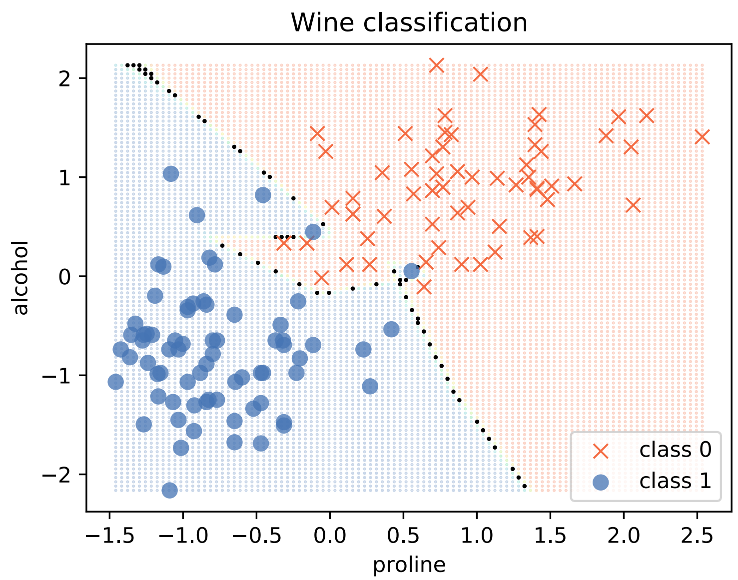


Likely overfit

Even ReLUs can get curvilinear surfaces

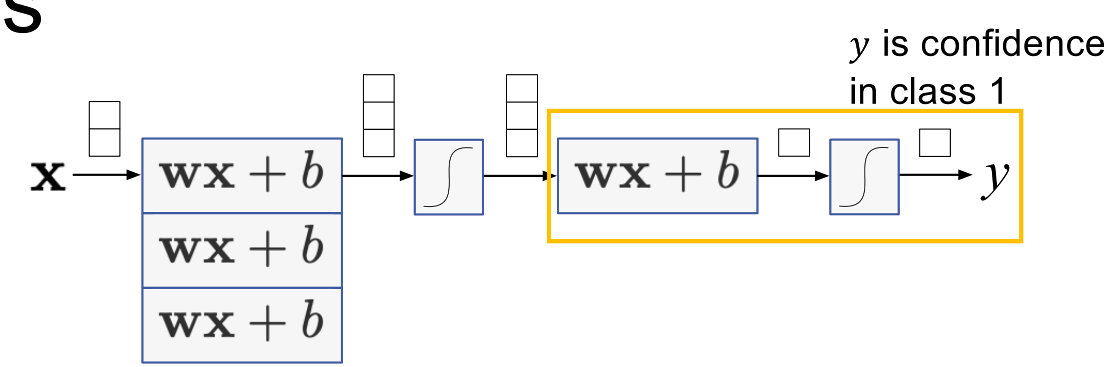
```
model = nn.Sequential(
    nn.Linear(2, 10),
    nn.ReLU(),
    nn.Linear(10, 10),
    nn.ReLU(),
    nn.Linear(10, 1),
    nn.Sigmoid()
)
```

(Last activation function still must be sigmoid)

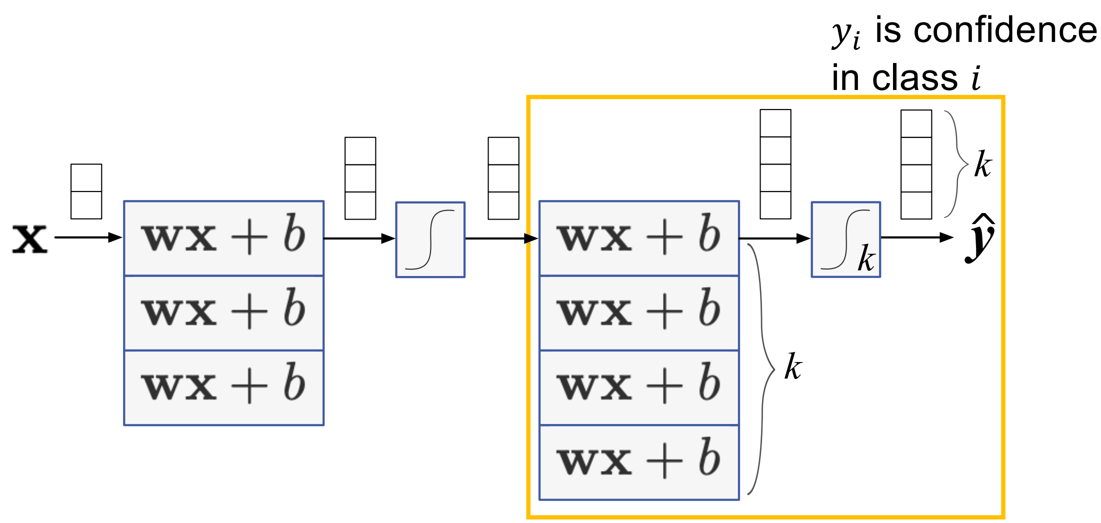


Multi-class classifiers

- **2-class problems**: final 1 neuron linear layer + sigmoid layer



- ***k*-class problems**: final k-neuron linear layer + softmax



k-class classifiers

- Instead of sigmoid, we use softmax function
- Instead of one neuron in last layer, we use k for k classes
- Last layer output: $\boldsymbol{z}^{[layer]} = W^{[layer]}\boldsymbol{x}^{\boldsymbol{T}} + \boldsymbol{b}^{[layer]}$
- Vector of k probabilities as activation: $\boldsymbol{y} = softmax\left(\boldsymbol{z}^{[layer]}\right)$
- Normalized probabilities of k class

$$softmax(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$$

Sample softmax computation

- For layer output vector $\boldsymbol{z}$: $$softmax(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$$

```
z = np.array([0.1, 1, 5])
np.exp(z)
```

```
array([  1.10517092,   2.71828183, 148.4131591 ])
```

```
np.exp(z) / np.sum(np.exp(z))
```

```
array([0.00725956, 0.01785564, 0.9748848 ])
```

Training deep learning networks

What does training mean?

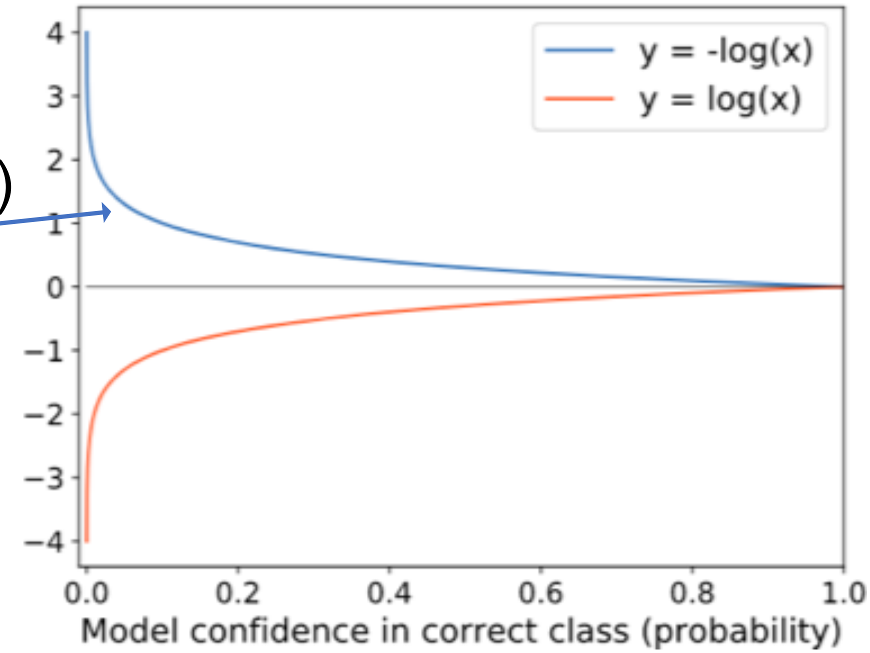
- Recall: testing an instance, a feature vector, means running that vector through the network and getting a prediction
- Prediction means computing a value using the model parameters; e.g., $\hat{y} = 3x + 2$ is a different model than $\hat{y} = .5x + 10$
- Find optimal (or good enough) model parameters as measured by a *loss* (cost) function
- Loss function measures the difference between model predictions and known targets
- We have huge search space (of parameters) and it is challenging to find parameters giving low loss

Loss functions

- **Regression**: typically mean squared error (MSE); should have smooth derivative, though mean absolute error works despite discontinuity (it's derivative is a V shape)
- **Classification**: log loss (also called cross entropy)
  - Penalizes very confident misclassifications strongly
  - Function of actual y and estimated probabilities, not predicted class
  - Perfect score is 0 log loss, imperfection gives unbounded scores
  - PyTorch log loss: `loss = cross_entropy(y_softmax, y_true)`
  - Predictions: `y_pred = argmax(y_softmax)`

Log loss

- loss = *penalty*(p) if y=1 else *penalty*(1– p)
- Let *penalty*(p) = -log(p)



- Two-class log loss:

$$loss = -\frac{1}{n}\sum_{i=1}^{n} y_i \log(p) + (1 - y_i)\log(1 - p_i)$$

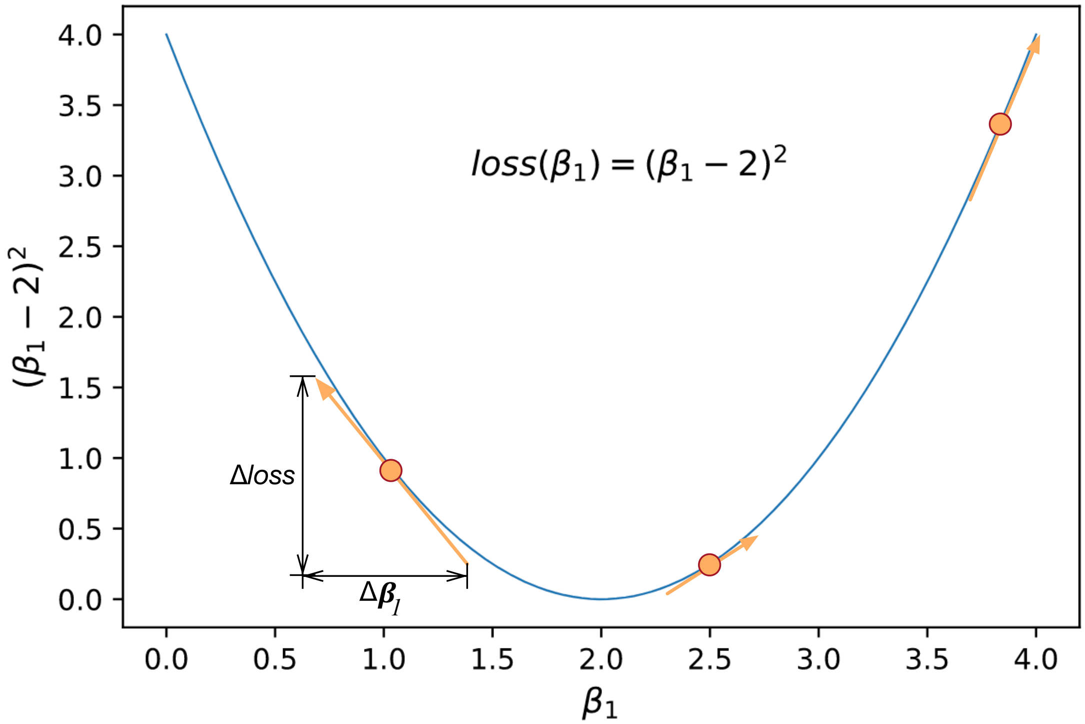
So log loss is average penalty where penalty is very high for confidence in wrong answer

UNIVERSITY OF SAN FRANCISCO

Minimize loss with Gradient descent

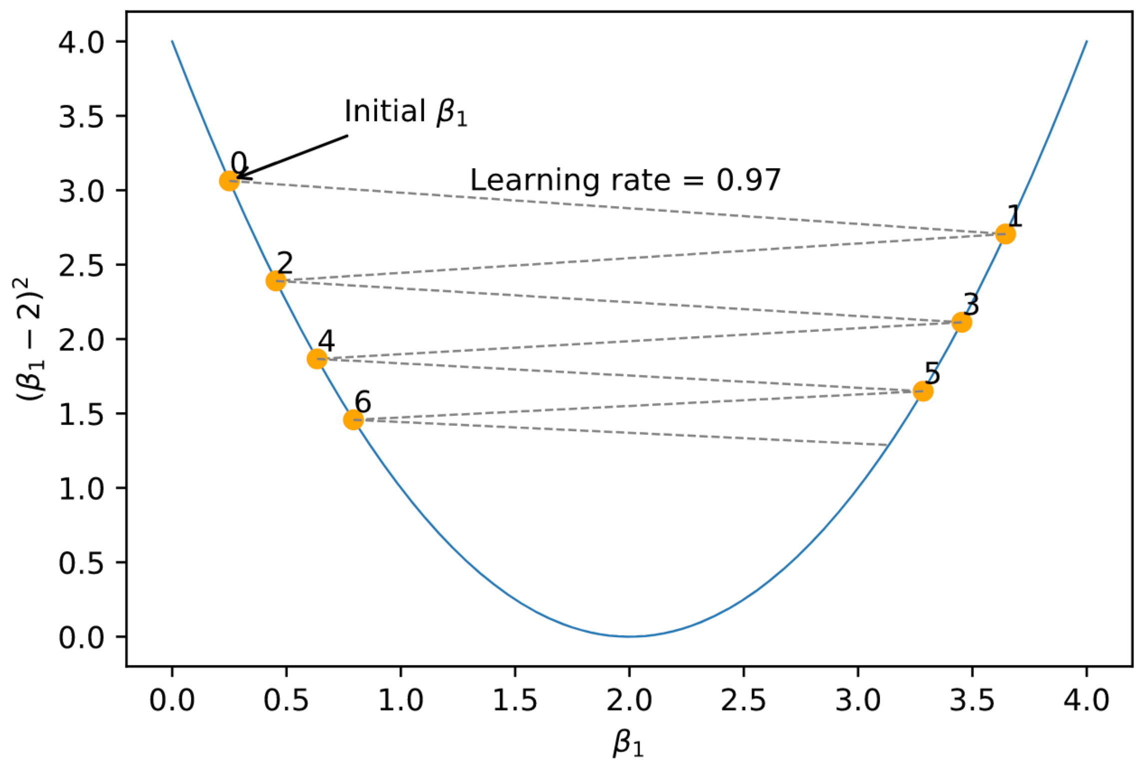
```
while not_converged:
    β = β - rate * gradient(β)
```

- We use information about the loss function in the neighborhood of current parameters (here called β_i) to decide which direction shifts towards smaller loss
- Tweak parameters in that direction, amplified by a learning rate



If learning rate is too high?

- We oscillate across valleys
- It can even diverge, exploding
- If too small, we don't make progress to min



UNIVERSITY OF SAN FRANCISCO

Training process

1. Prepare data, normalize numeric variables
2. Split out at least a validation set from training set
3. Choose network architecture, appropriate loss function
4. Choose hyper-parameters, such as dropout rate
5. Choose a learning rate, number of epochs (passes through data)
6. Run training loop (until validation error goes up or num iterations)
7. Goto 3, 4, or 5 to tweak; iterate until good enough

Training loop

Regression

Vectorized forward network pass

```
for epoch in range(nepochs):
    y_train_pred = model(X_train)
    loss = MSE(y_train, y_train_pred) # assume final sigmoid
    update model parameters in direction of lower loss
```

Classification

```
for epoch in range(nepochs):
    y_train_pred = model(X_train)
    loss = cross_entropy(y_train, y_train_pred) # assume softmax
    update model parameters in direction of lower loss
```

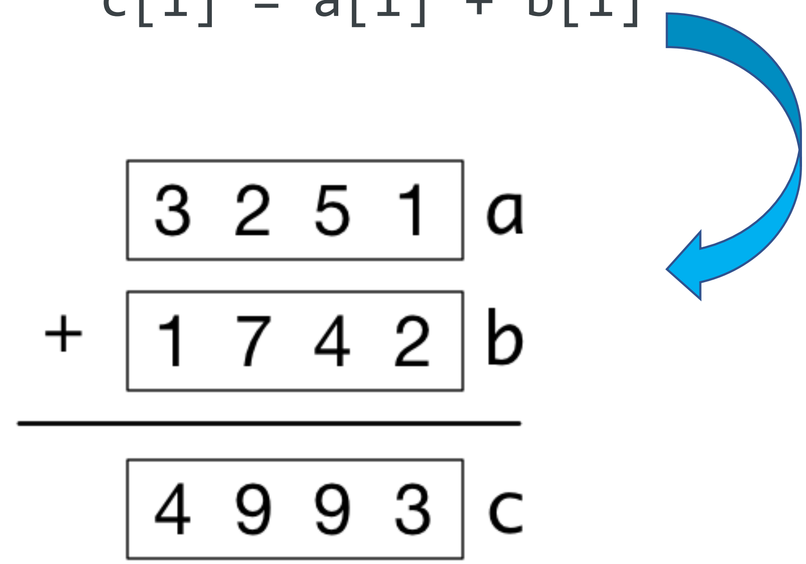
What is vectorization?

- Use vectors not loops
- For torch/numpy arrays, we can use vector math instead of a loop:

```
c = a + b
```

- Gives an opportunity to execute vector addition in parallel

```
for i in range(len(a)):
    c[i] = a[i] + b[i]
```

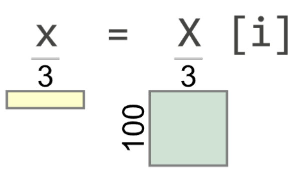


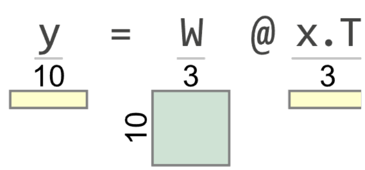
Vectorization in training loop

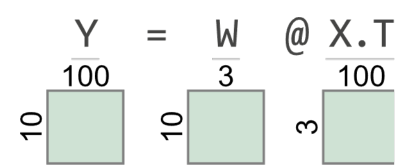
- Running one instance through network is how we think about it
- In practice, we send a subset or all X instances through the network in one go and compare all predictions to all $\boldsymbol{y}$
- Instead of looping through instances, we pass all of X through to use matrix-matrix multiplies instead of matrix-vector multiplies

```
for epoch in range(nepochs):
    for i in range(n):
        y = model(X[i])
        …
```

```
for epoch in range(nepochs):
    Y = model(X)
    …
```



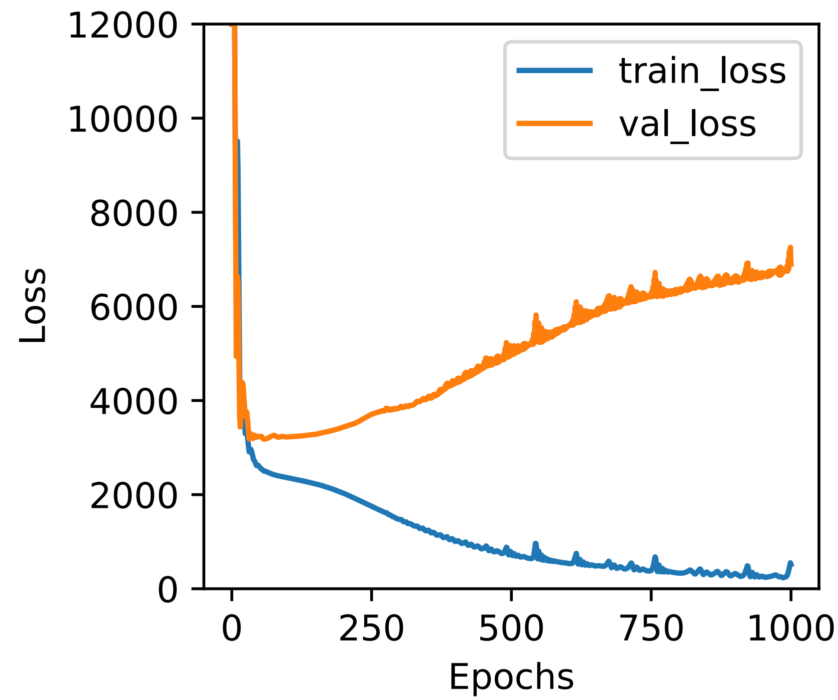




Assume n=100, m=3, n_neurons=10 in weight matrix W

Common train vs validation loss behavior

- DL networks have so many parameters, we can often get training error down to zero!
- But, we care about generalization
- Unfortunately, validation error often tracks away from training error as the number of epochs increases
- This model is clearly overfitting
- Need to use regularization to improve validation loss



Regularization techniques

- Get more training data; can try augmentation techniques
  (more data is likely to represent population distribution better)
- Reduce number of model parameters (i.e., simplify it)
  (not powerful enough to fit the noise)
- Add drop out layers (randomly kill some neurons)
- Weight decay (L2 regularization on model parameters)
- Early stopping, when validation error starts to go up
  (generally we choose model that yields the best validation error)
- Batch normalization has some small regularization effect
  (Force layer activation distributions to be 0-mean, variance 1)
- Stochastic gradient descent tends to land on better generalizations