Fundamentals of deep learning

Crash course in using PyTorch to train models

Terence Parr
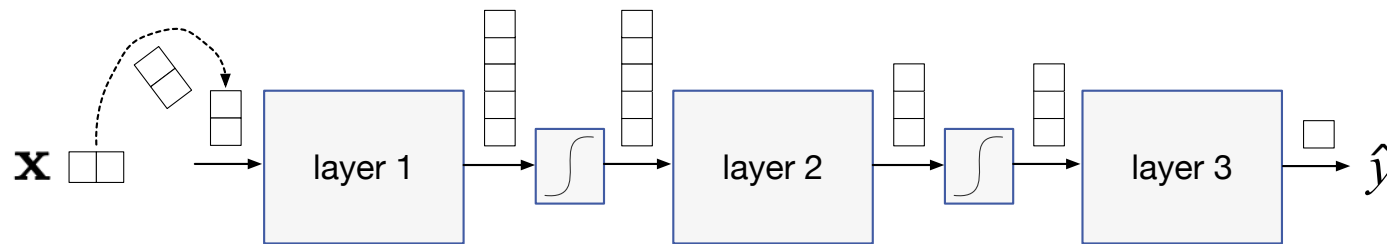MSDS program
University of San Francisco

Deep learning regressors

What's a neural network?

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

- Transform raw x vector into better and better features, final linear layer can then make excellent prediction
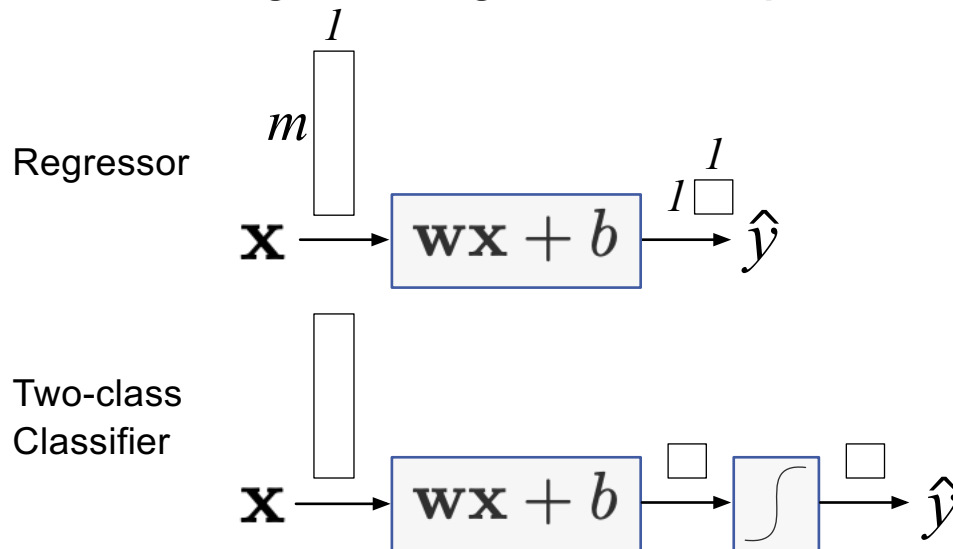
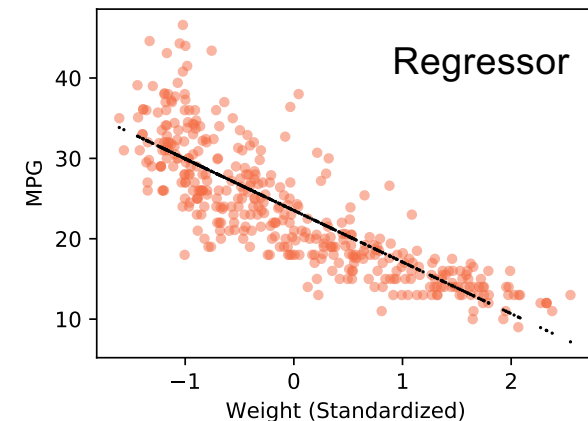# DL Building blocks

$$\mathbf{wx} + b$$ linear   sigmoid   ReLU (rectified linear unit)

- $\hat{y} = w_1 x_1 + w_2 x_2 + \ ... + w_m x_m + b = \boldsymbol{wx^T} + b$ for n x m dim X
- Linear/logistic regression equivalents (one $\boldsymbol{x}$ instance):

Regressor

$$\mathbf{x} \longrightarrow \boxed{\mathbf{wx} + b} \longrightarrow \hat{y}$$

Two-class
Classifier

$$\mathbf{x} \longrightarrow \boxed{\mathbf{wx} + b} \longrightarrow \boxed{\int} \longrightarrow \hat{y}$$

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

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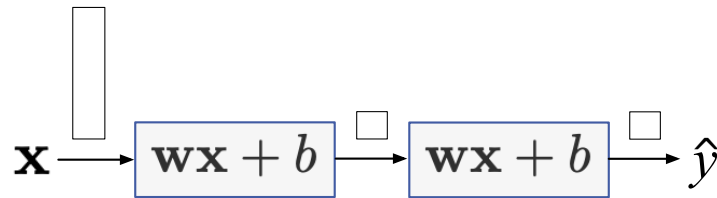
Regressor

Underfitting a bit here
(need more of a quadratic)

Try adding layers to get more power
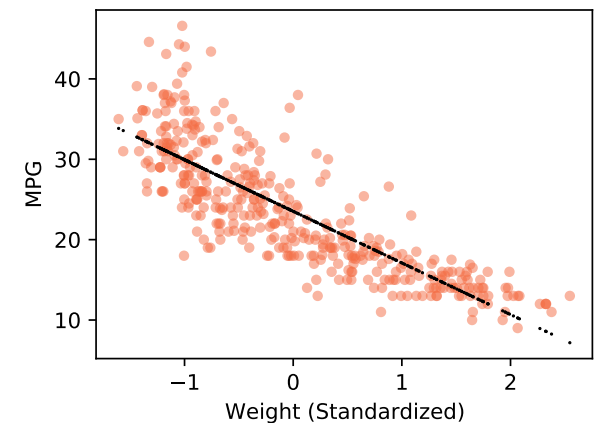
- But, sequence of linear models is just a linear model

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

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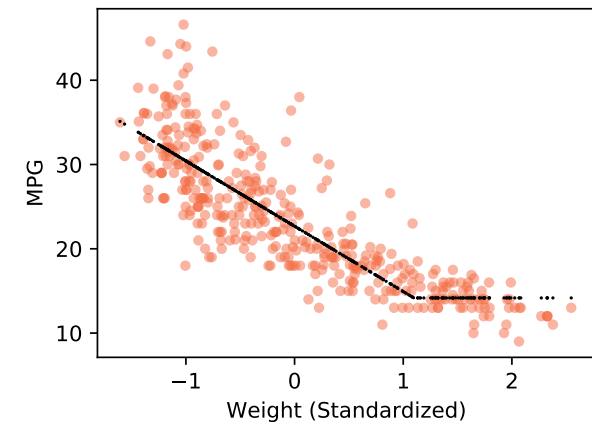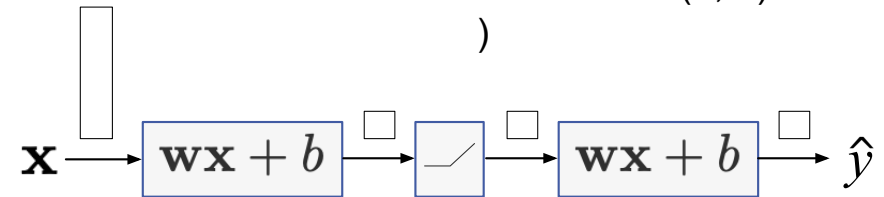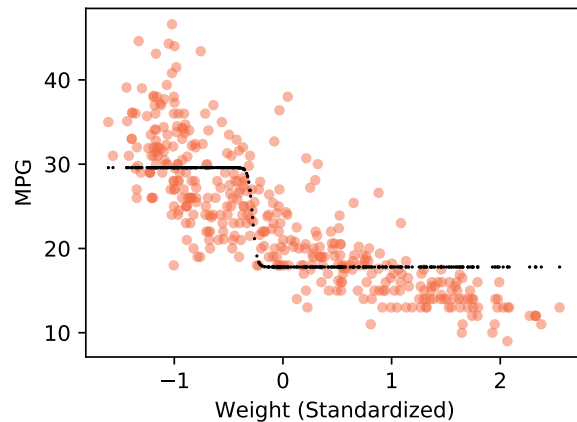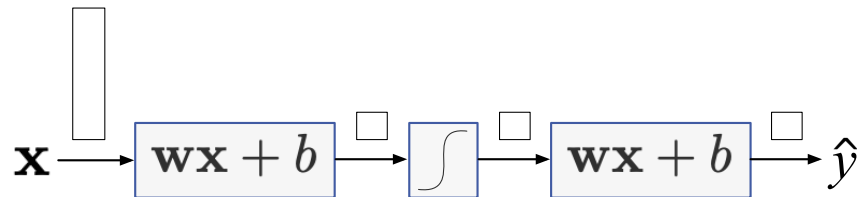

- PyTorch code

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


Still just a line

Must introduce nonlinearity


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

$$\mathbf{x} \longrightarrow \boxed{\mathbf{wx} + b} \longrightarrow \boxed{\int} \longrightarrow \boxed{\mathbf{wx} + b} \longrightarrow \hat{y}$$

$$\mathbf{x} \longrightarrow \boxed{\mathbf{wx} + b} \longrightarrow \boxed{\diagup} \longrightarrow \boxed{\mathbf{wx} + b} \longrightarrow \hat{y}$$

ReLU idea here: Draw two lines
then clip at intersection

Stack linear models (neurons) for more power

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


```
model = nn.Sequential(
    nn.Linear(m, 5),
    nn.ReLU(),
    nn.Linear(5, 1)
)
```
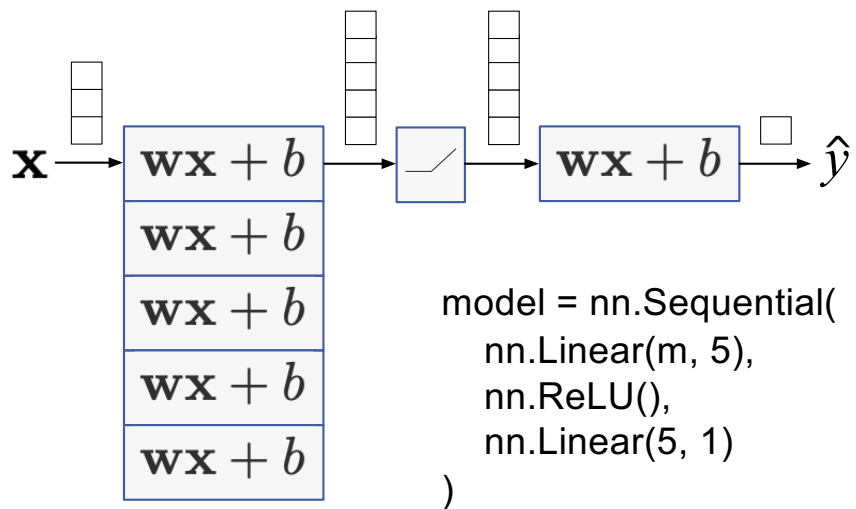
All those $\boldsymbol{w}$ and b are different $W^{[1]}$ means W for layer 1

UNIVERSITY OF SAN FRANCISCO
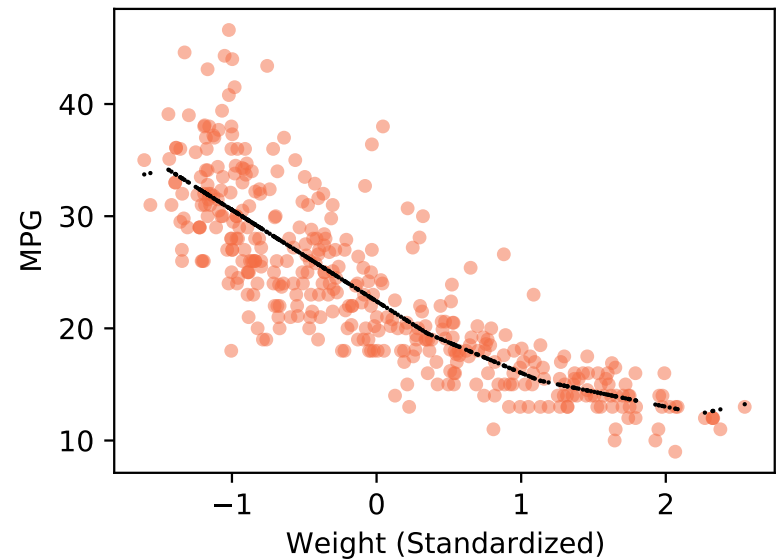
Math for dataset 1D: weight→MPG

$W^{[1]}$ $W^{[2]}$ $W^{[3]}$

$$\mathbf{x} \rightarrow \boxed{\begin{matrix}\mathbf{wx}+b\\\mathbf{wx}+b\\\mathbf{wx}+b\\\mathbf{wx}+b\\\mathbf{wx}+b\end{matrix}} \rightarrow \boxed{/} \xrightarrow{a^{[1]}} \boxed{\begin{matrix}\mathbf{wx}+b\\\mathbf{wx}+b\end{matrix}} \rightarrow \boxed{/} \xrightarrow{a^{[2]}} \boxed{\mathbf{wx}+b} \xrightarrow{a^{[3]}} \hat{y}$$

(leaving out b's)

```
a1 = F.relu(W1 @ x)
```

```
a2 = F.relu( W2 @ a1)
```

```
a3 = W3 @ a2
```
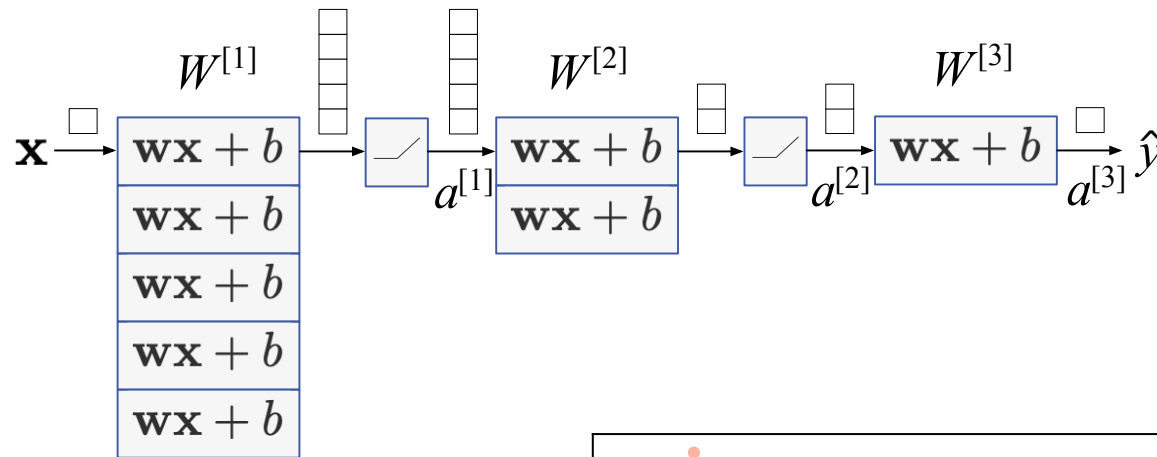
```
model = nn.Sequential(
    nn.Linear(1, 5),
    nn.ReLU(),
    nn.Linear(5, 2),
    nn.ReLU(),
    nn.Linear(2, 1)
)
```

(courtesy of TensorSensor)
https://explained.ai/tensor-sensor/index.html

UNIVERSITY OF SAN FRANCISCO

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)
)
```


UNIVERSITY OF SAN FRANCISCO

Classifiers

UNIVERSITY OF SAN FRANCISCO
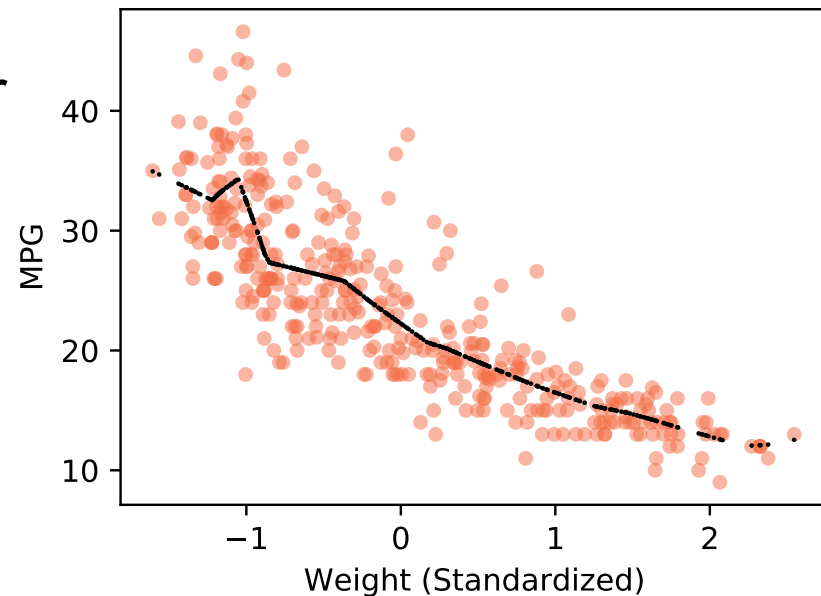
Binary classifiers

- Add sigmoid to regressor and we get a two-class classifier

- Prediction $\hat{y}$ is probability of class 1

- One-layer (hidden) network with sigmoid activation function is just a logistic regression model

- Provides hyper-plane decision surfaces

Wine classification

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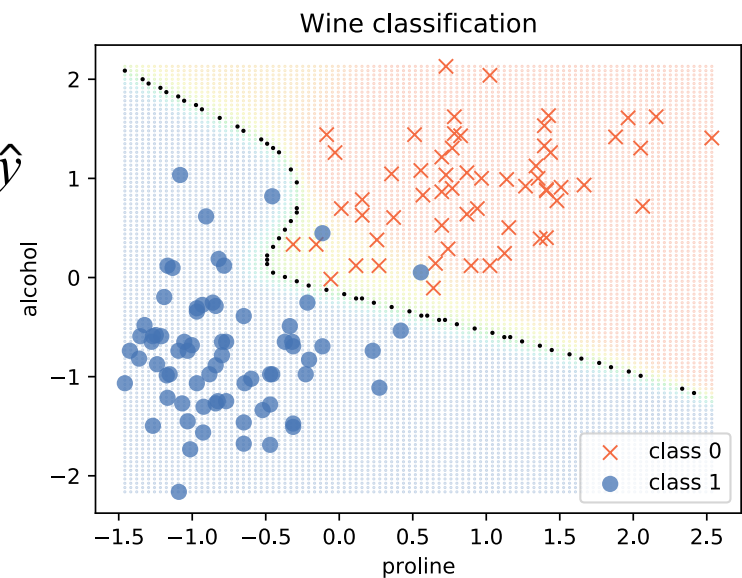```

UNIVERSITY OF SAN FRANCISCO
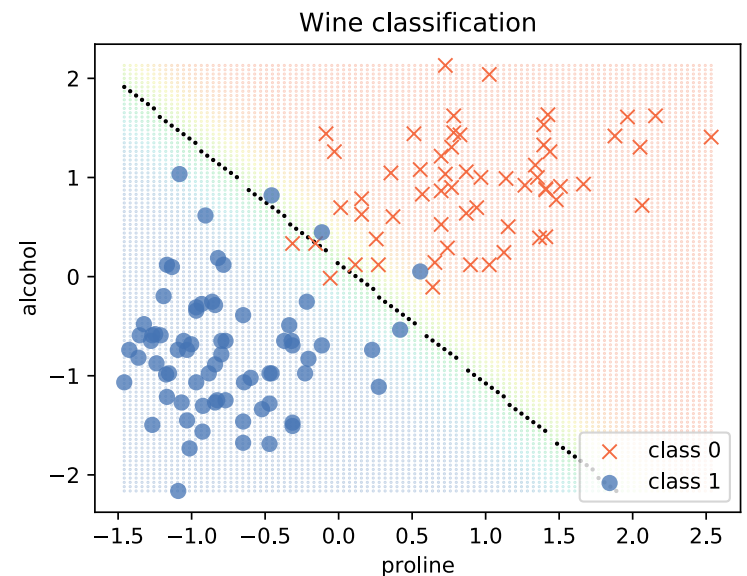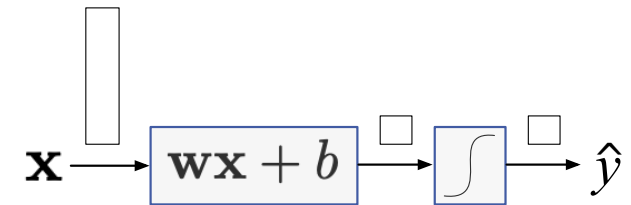
# Stack neurons, add layer

- We get a nonlinear decision surface
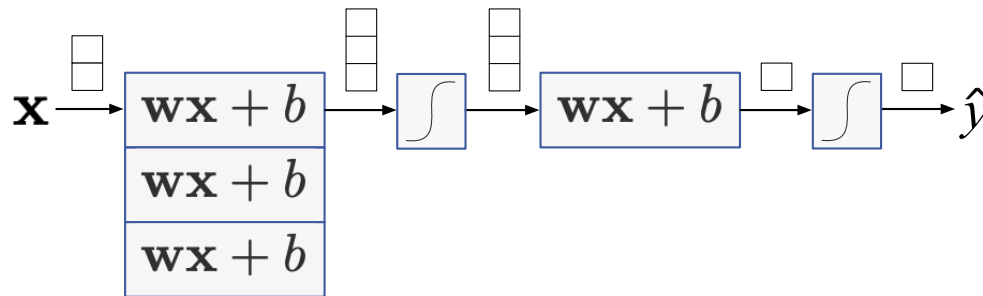


```
model = nn.Sequential(
    nn.Linear(2, 3),
    nn.Sigmoid(),
    nn.Linear(3, 1),
    nn.Sigmoid()
)
```

All those w and b are different

UNIVERSITY OF SAN FRANCISCO

More neurons:
more complex decision surface

$$\mathbf{x} \rightarrow \boxed{\mathbf{wx}+b} \rightarrow \boxed{\int} \rightarrow \boxed{\mathbf{wx}+b} \rightarrow \boxed{\int} \rightarrow \hat{y}$$

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

Wine classification
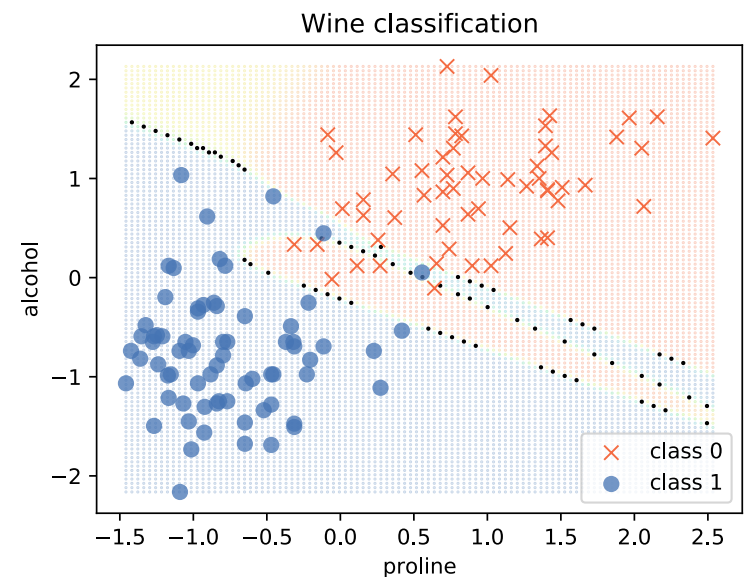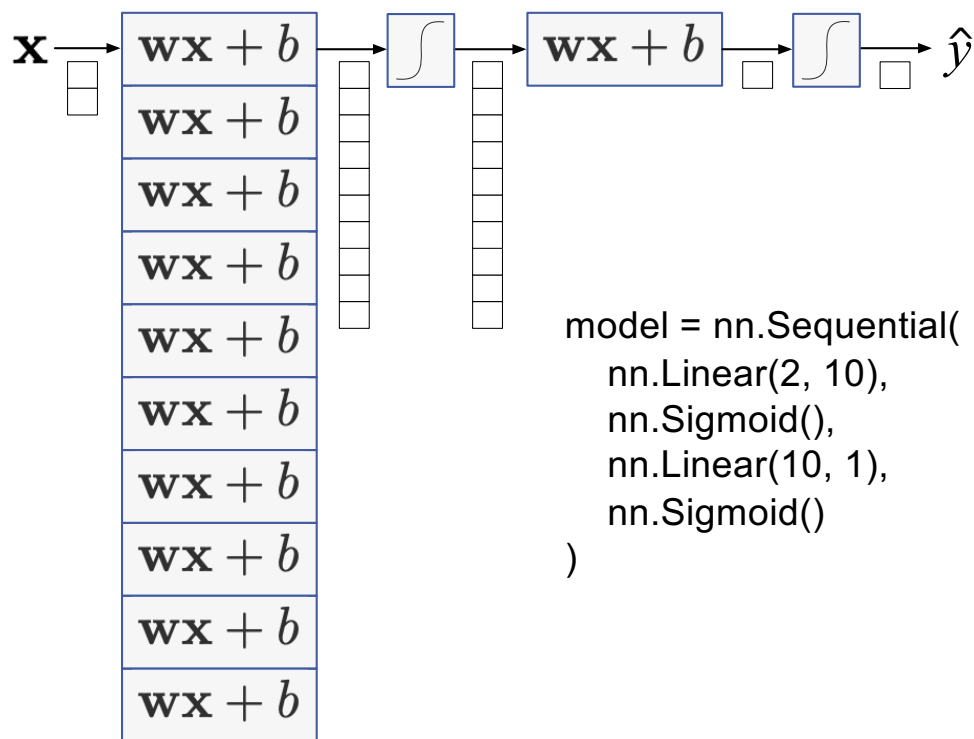
Likely overfit

Not only more complex than hyperplane but non-contiguous regions!

Even ReLUs can get "curvy" 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)

UNIVERSITY OF SAN FRANCISCO
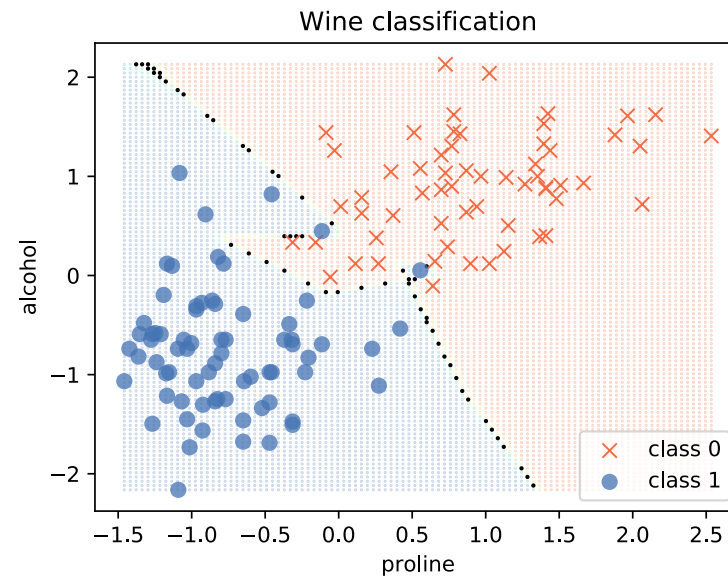
k-class classifiers

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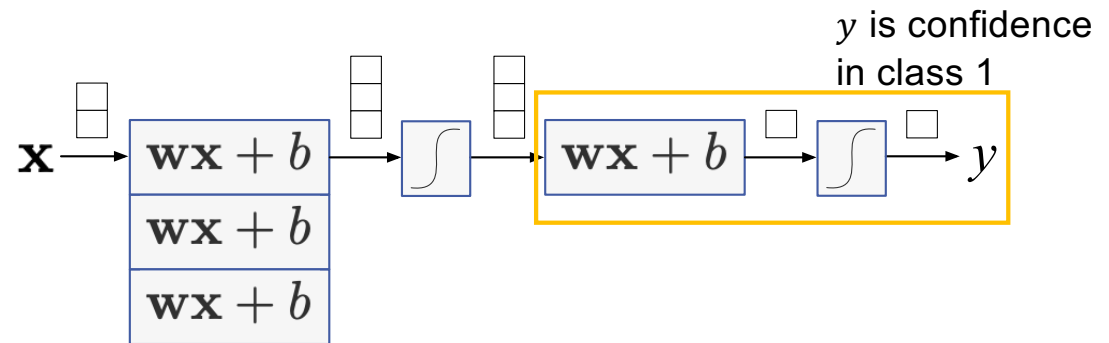
- **k-class problems**: final k-neuron linear layer + softmax

y is confidence in class 1

y_i is confidence in class i
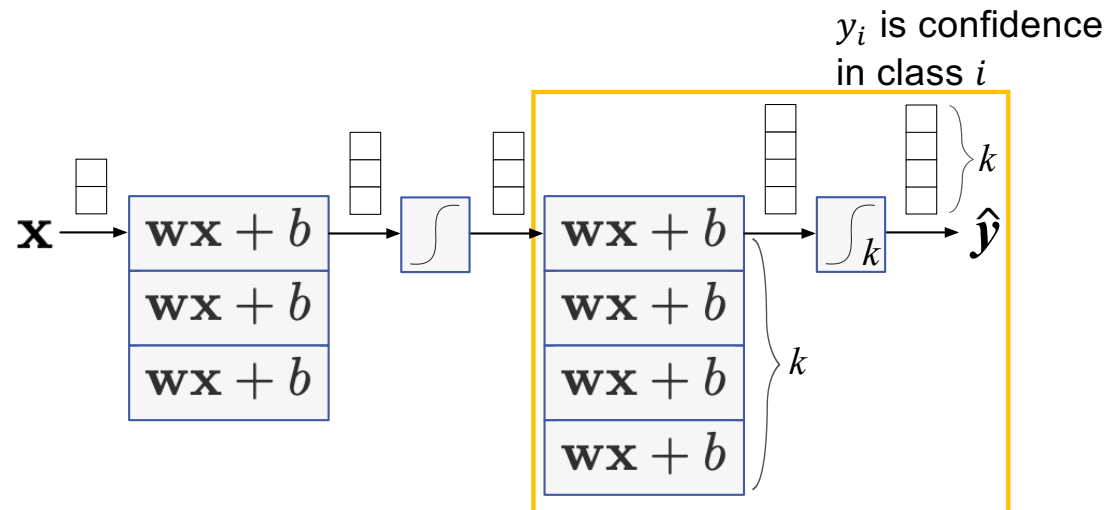
k-class classifiers

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

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

Sample softmax computation

- For layer output vector **z**:

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

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

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

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

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

Training deep learning networks

Refresher: What does training mean?

- Making a prediction means running a feature vector through the network
  - i.e., computing a value using the model parameters; e.g., $\hat{y} = 3x + 2$ is a different model than $\hat{y} = .5x + 10$

- Training: 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

UNIVERSITY OF SAN FRANCISCO

Refresher: 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

penalty(p)

- Let p be predicted probability that y=1
- loss = *penalty*(p) if y=1 else *penalty*(1– p)
- Let *penalty*(p) = -log(p)
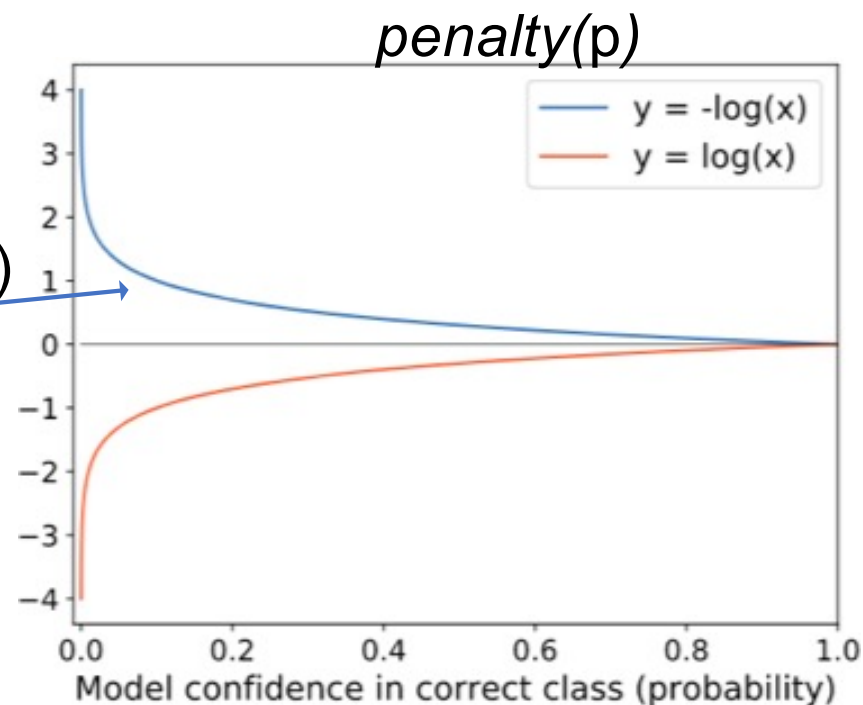
- Two-class log loss:
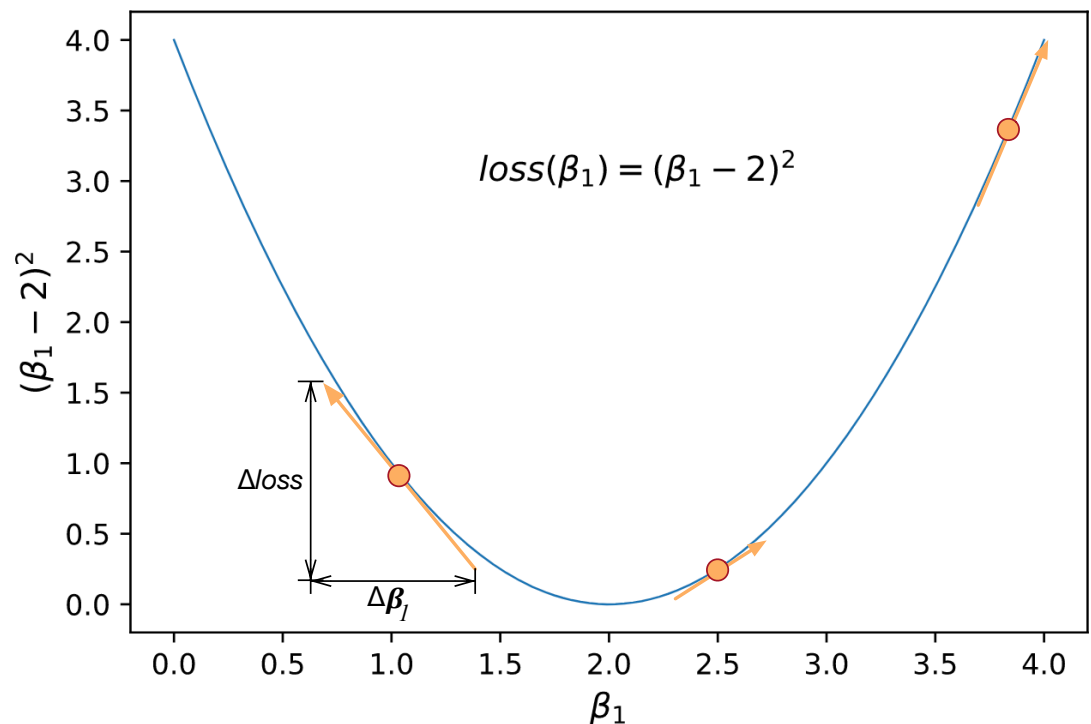
$$loss = -\frac{1}{n}\sum_{i=1}^{n} y_i\log(p) + (1 - y_i)\log(1 - p_i)$$

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
- Go in opposite dir of slope

$$loss(\beta_1) = (\beta_1 - 2)^2$$

y-axis: $(\beta_1 - 2)^2$, x-axis: β_1, with $\Delta loss$ and $\Delta \beta_1$ labeled

UNIVERSITY OF SAN FRANCISCO

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
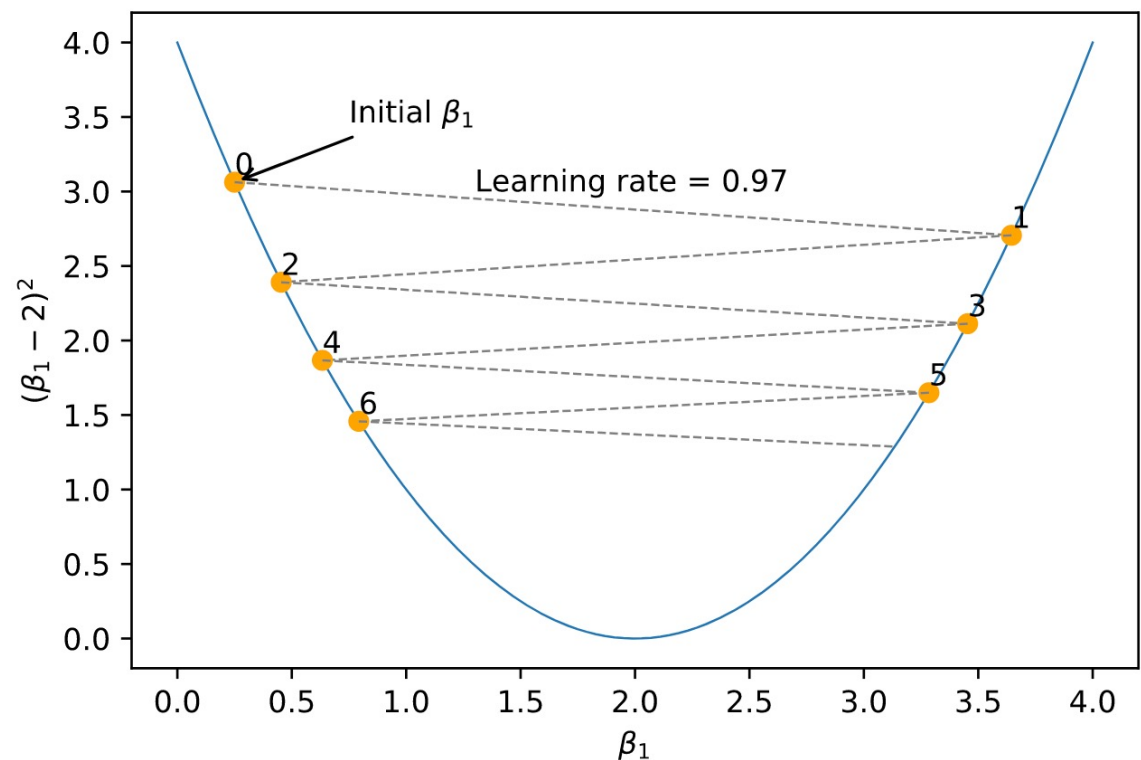
Training process

1. Prepare data
   - normalize numeric variables
   - dummy vars for categoricals
   - conjure up values for missing values
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

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

Vectorized forward network pass
(send in all instances at once)

Classification

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

UNIVERSITY OF SAN FRANCISCO

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
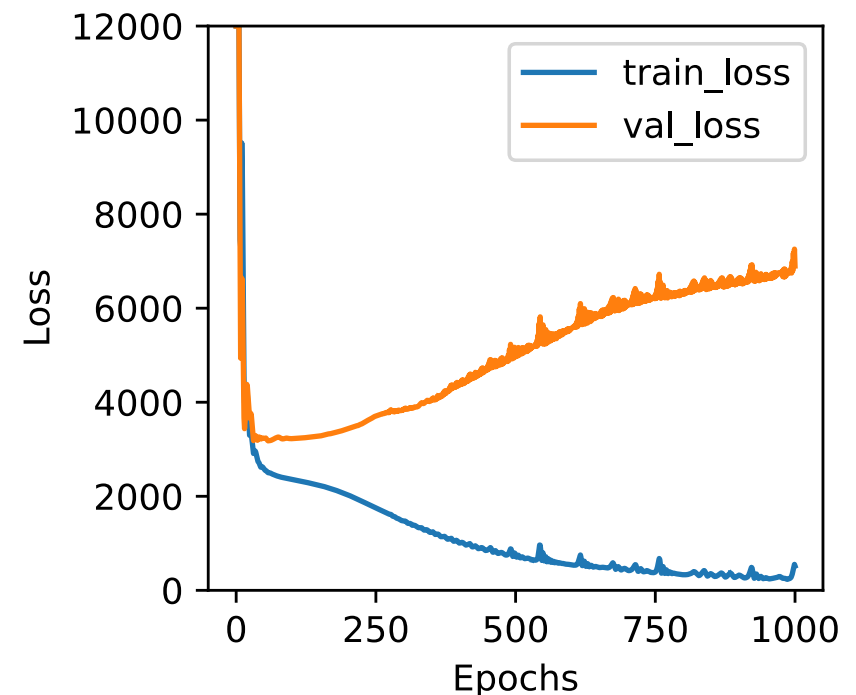
Regularization techniques

- 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) (reduce power/ability to fit the noise)

- Add drop out layers (randomly kill some neurons)

- Weight decay (L2 regularization on model parameters, restrict model parameter search space)

- 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

Aside: 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 $\hat{y}$ predictions to all y

- Instead of looping through instances, we pass X through to use matrix-matrix multiplies instead of matrix-vector multiplies

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



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


Get 100 answers

Assume n=100, m=3, n_neurons=1 in 1x3 weight matrix W

UNIVERSITY OF SAN FRANCISCO

Summary

- Vanilla deep learning models are layers of linear regression models glued together with nonlinear functions such as sigmoid/ReLUs

- Regressor: final layer transforms previous layer to single output

- Classifier: add sigmoid to last regressor layer (2-class) or add softmax to last layer of k neurons (k-class)

- Training a model means finding optimal (or good enough) model parameters as measured by a *loss* (cost or error) function; hyper parameters describe architecture and learning rate, amount of regularization, etc.

- We train using (stochastic) gradient descent; tuning model and hyper parameters is more or less trial and error ☹ but experience helps a lot