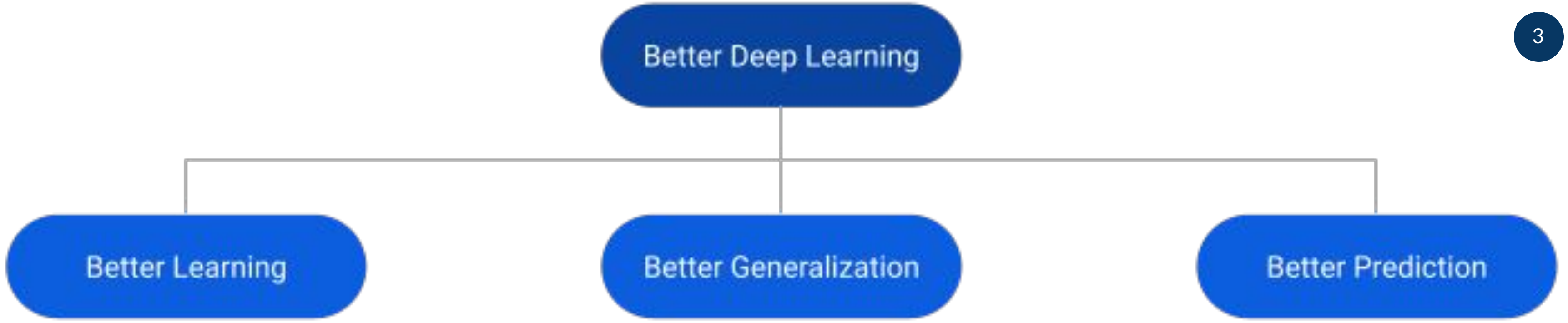
Better Deep Learning

Better Generalization vs Better Learning



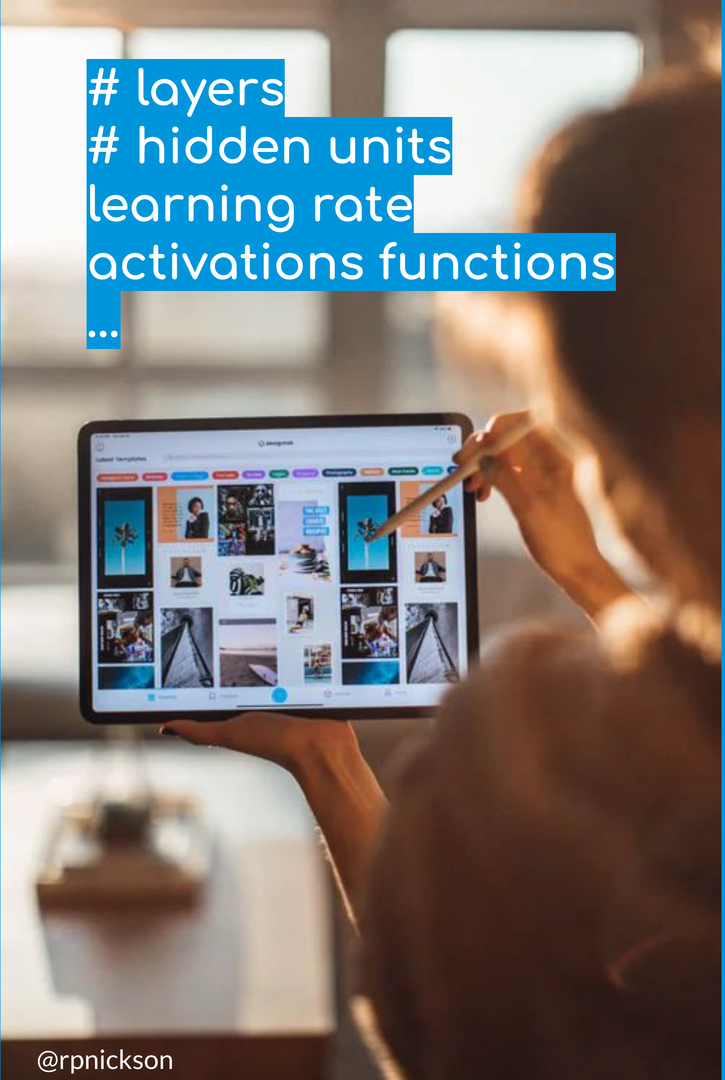
1. **Configure capacity with node and layers**
2. **Configure gradient precision with batch size**
3. **Configure what to optimize with loss functions**
4. **Configure speed of learning with learning rate**
5. Stabilize learning with data scaling
6. Fix vanishing gradient with relu
7. Fix exploding gradient with gradient clipping

1. Fix overfitting with regularization
2. Penalize large weights with weight regularization
3. Force small weights with weight constraint
4. Decouple layers with dropout
5. Promote robustness with noise
6. Halt training at the right time with early stopping

Neural Networks in Practice #01

Splitting Data

# layers
# hidden units
learning rate
activations functions
…

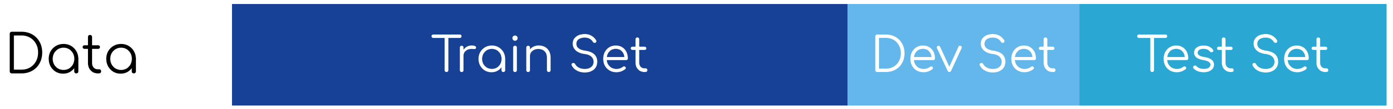


Applied NN is a highly iterative process

$$\begin{tikzcd} \text{Idea} \arrow[d] \\ \text{Code} \arrow[d] \\ \text{Experiment} \arrow[uu, bend right] \end{tikzcd}$$

Train - Dev - Test Sets

Making good choices in how you set up your training, development, and test sets can make a huge difference in helping you quickly find a good high performance neural network.



Holdout
Validation
Development

Previous ML era

- 70/30
- 60/20/20

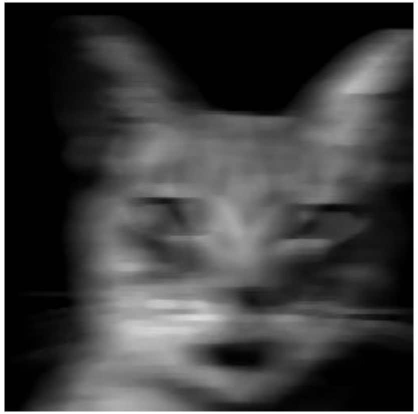
Big Data era

- 98/1/1
- 99.5/0.25/0.25
- 99.5/0.4/0.1

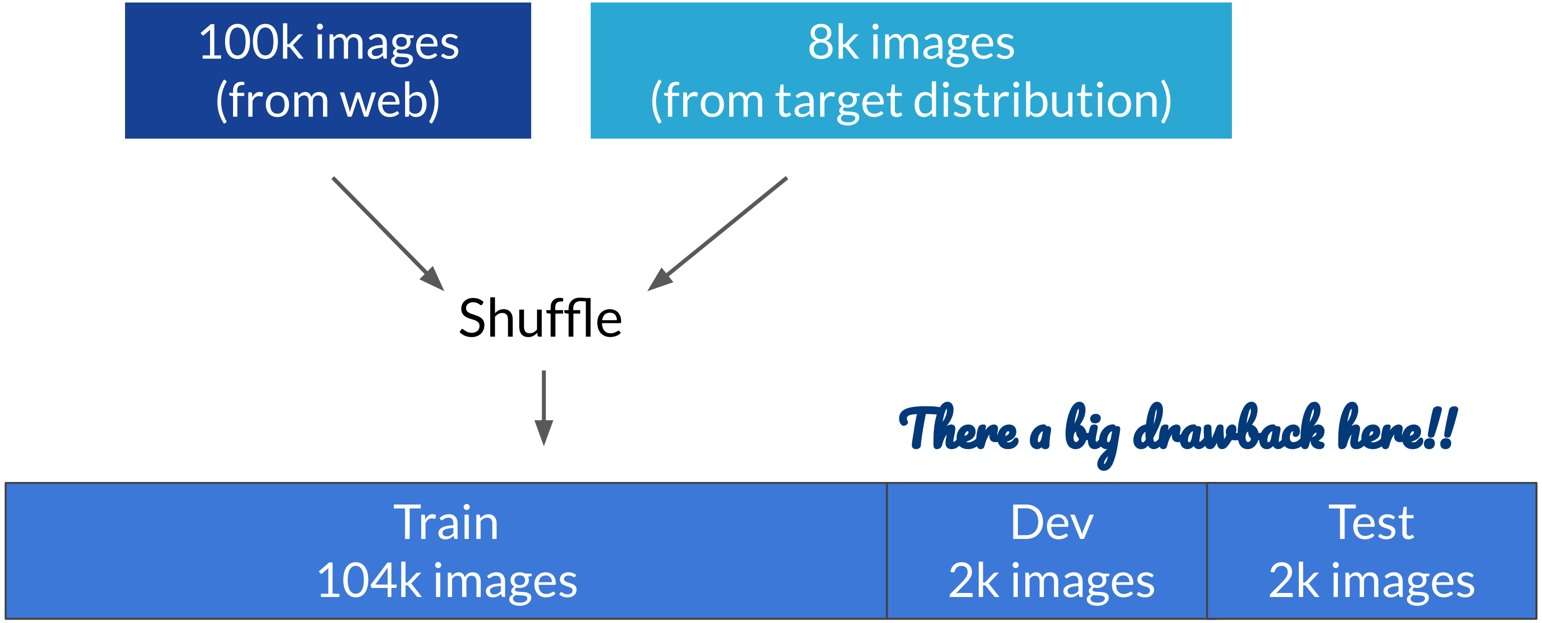
Mismatched train/test distribution

Scenario: say you are building a cat-image classifier application that determines if an image is of a cat or not. The application is intended for users in rural areas who can take pictures of animals by their mobile devices for the application to classify the animals for them.

Scraped from Web Pages
100k images

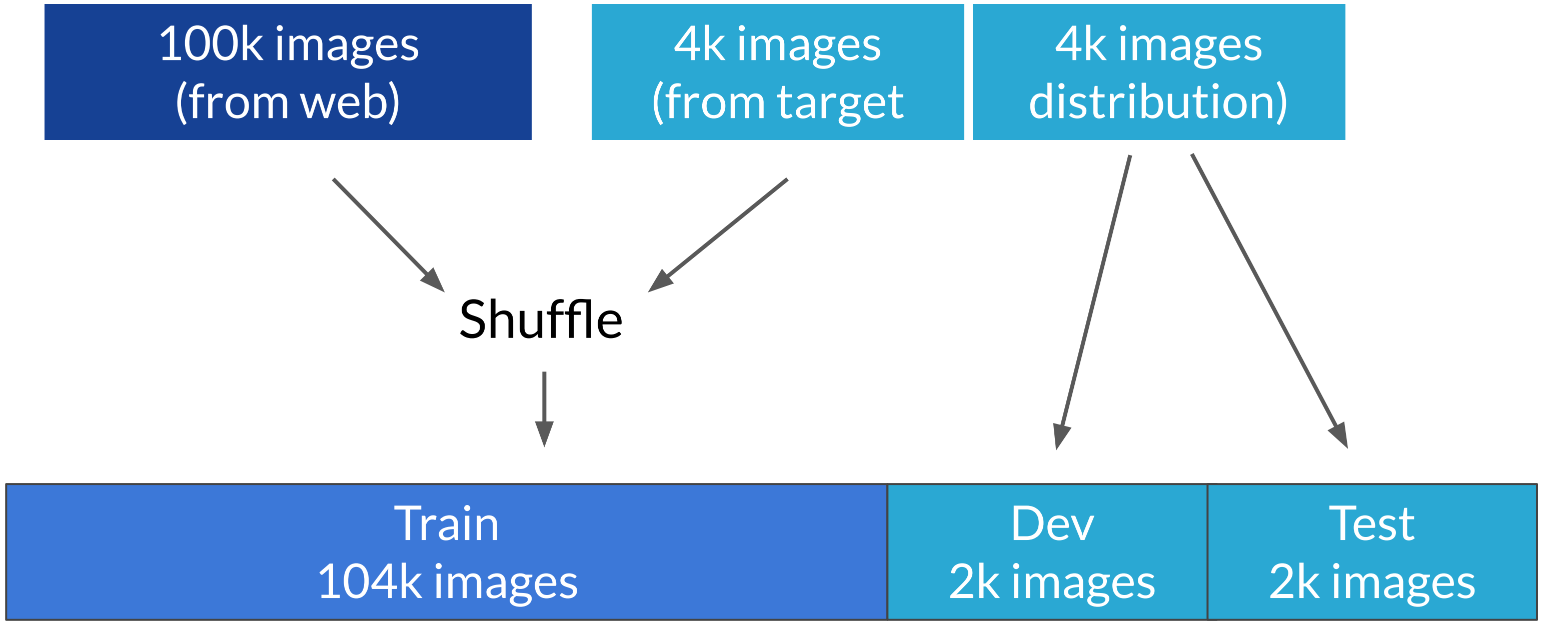

Collected from Mobile Devices
<<target distribution>>
8k images

A possible option: shuffling the data



only 148 images come from the target distribution

A better option

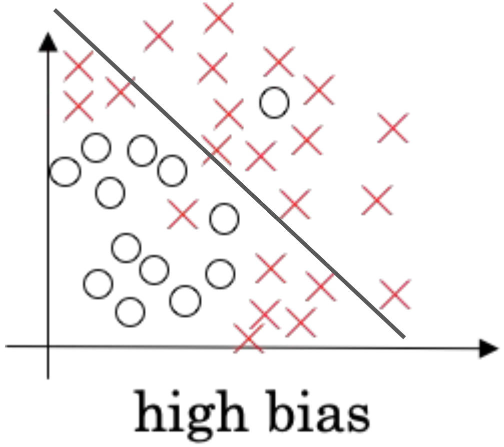


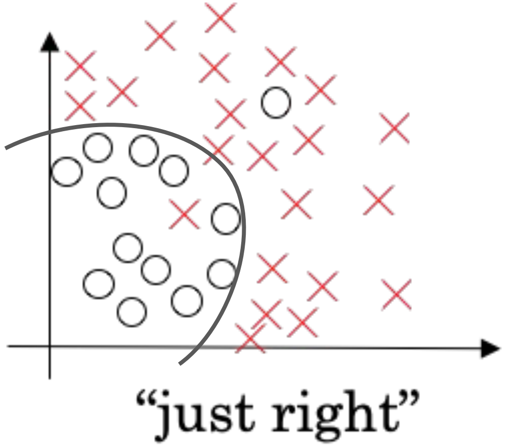
Rule of the thumb

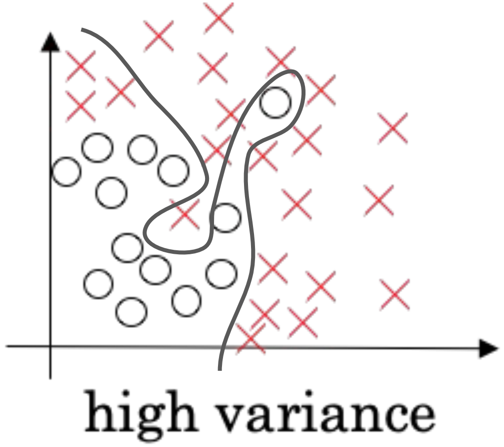
>> make sure that the dev and test sets come from the same distribution

Not having a test set might be okay. (Only dev set)

Bias vs Variance







Underfitting

Overfitting

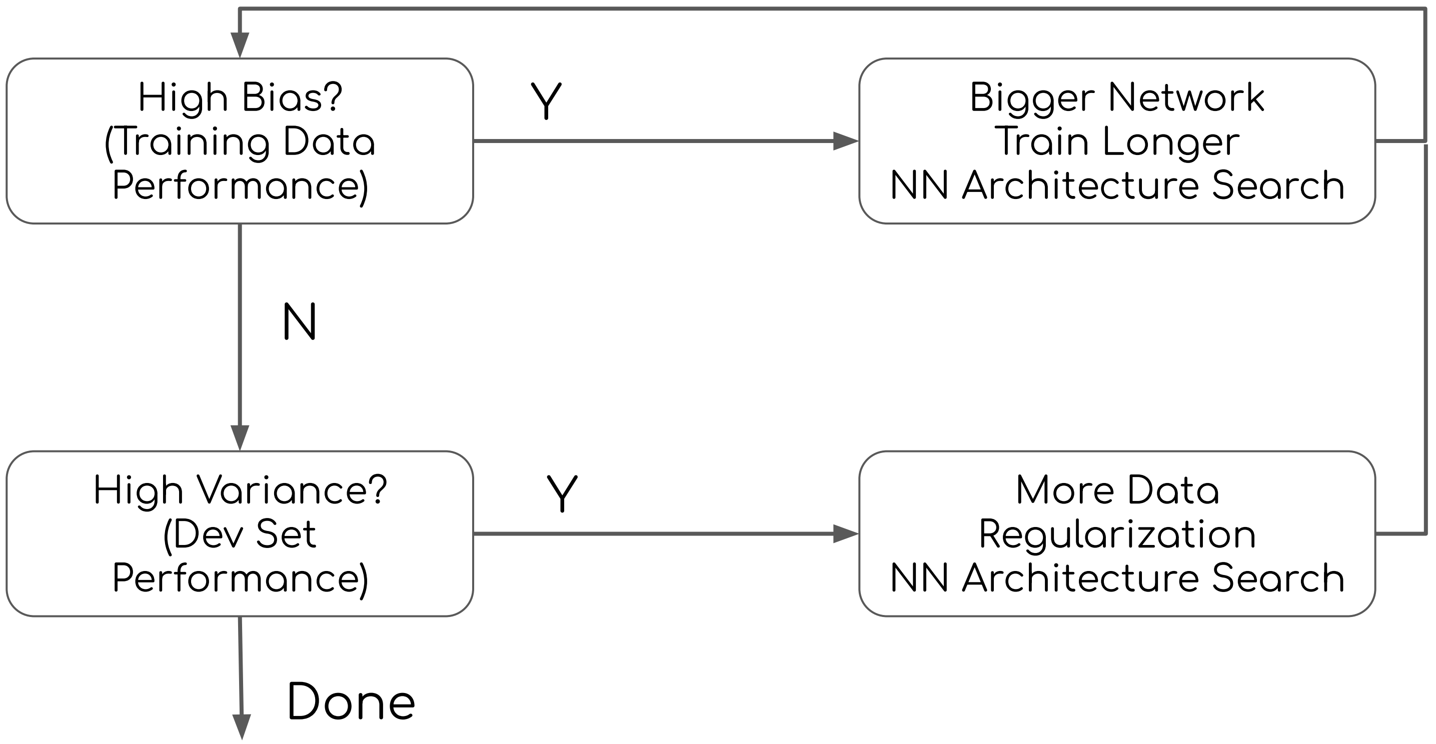
Bias vs Variance

Cat Classification

	Scenario #01	Scenario #02	Scenario #03	Scenario #04
Train Set Error	1%	15%	15%	0.5%
Dev Set Error	16%	16%	30%	1%

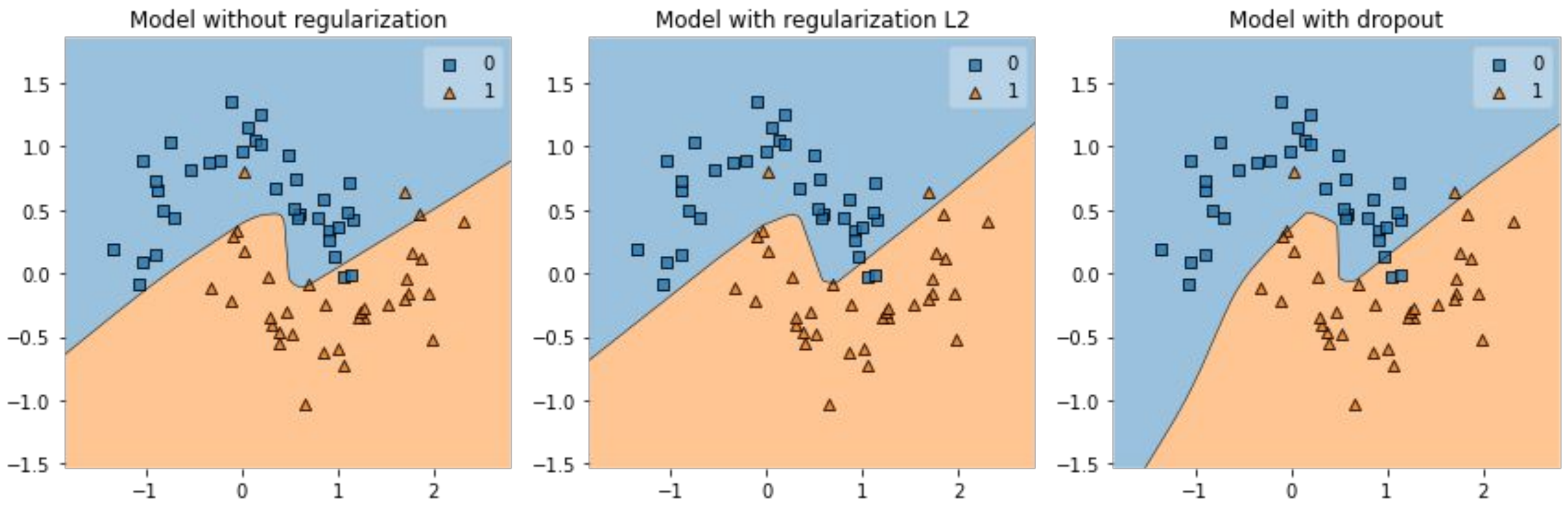
Low Bias High Variance	High Bias Low Variance	High Bias High Variance	Low Bias Low Variance

Basic Recipe for Machine Learning

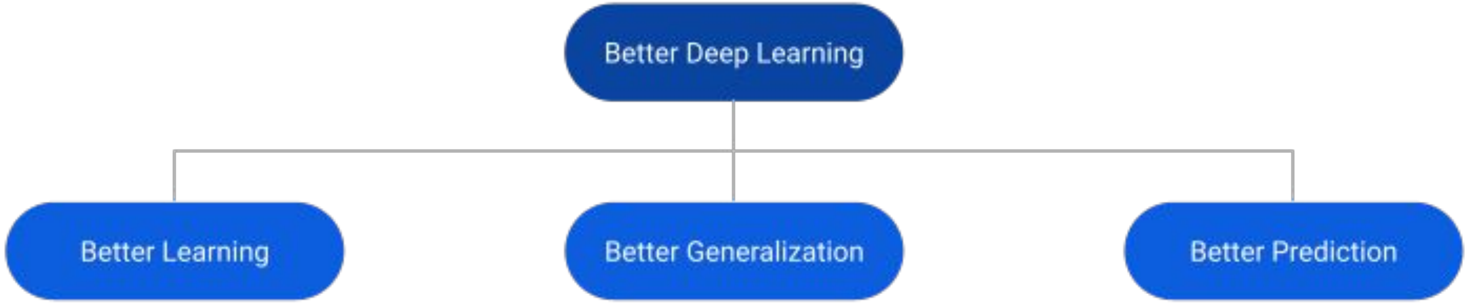


Regularizing your Neural Network

What if we penalize complexity?



It is very important that you regularize your model properly because it could dramatically improve your results

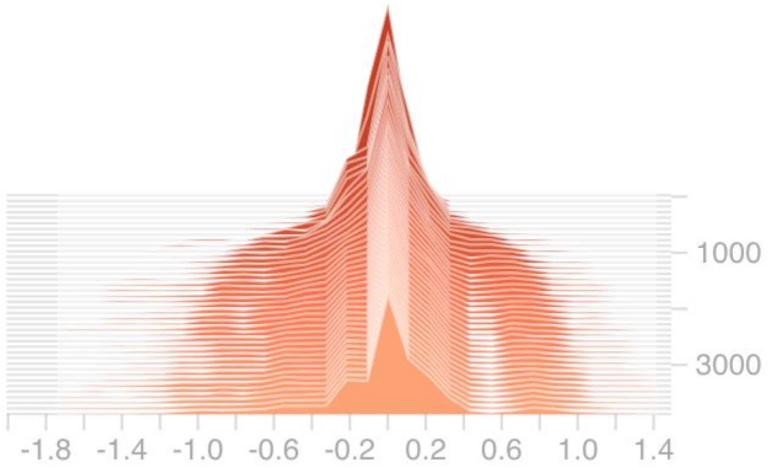


1. **Configure capacity with node and layers**
2. **Configure gradient precision with batch size**
3. **Configure what to optimize with loss functions**
4. **Configure speed of learning with learning rate**
5. Stabilize learning with data scaling
6. Fix vanishing gradient with relu
7. Fix exploding gradient with gradient clipping

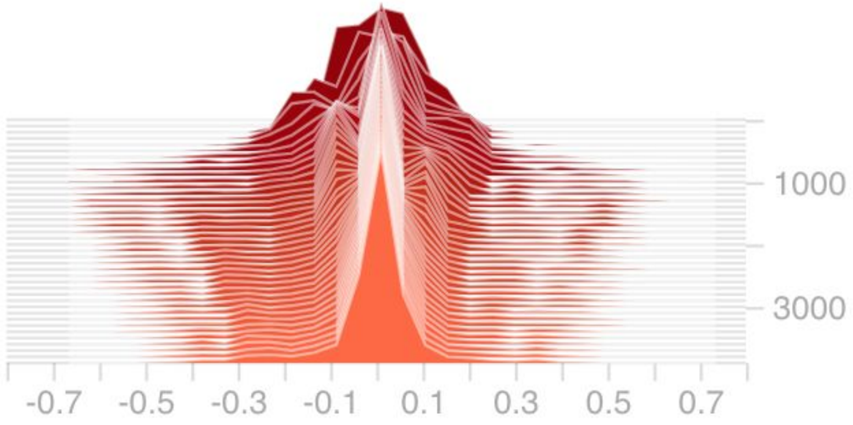
1. Fix overfitting with regularization
2. Penalize large weights with weight regularization
3. Force small weights with weight constraint
4. Decouple layers with dropout
5. Promote robustness with noise
6. Halt training at the right time with early stopping

Penalize Large Weights with Weight Regularization

Model without regularization



Model with weight regularization



https://www.tensorflow.org/tensorboard

L2 Regularization (Frobenius Norm)

$$J = -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(\hat{y}^{(i)}\right) + (1-y^{(i)})\log\left(1-\hat{y}^{(i)}\right)\right)$$

$$J_{regularized} = \underbrace{-\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(\hat{y}^{(i)}\right) + (1-y^{(i)})\log\left(1-\hat{y}^{(i)}\right)\right)}_{\text{cross-entropy cost}} + \underbrace{\frac{1}{m}\frac{\lambda}{2}\sum_{l}\sum_{k}\sum_{j}W_{k,j}^{[l]2}}_{\text{L2 regularization cost}}$$

L2 Regularization (Frobenius Norm)

Impact on Gradient Descent

$$J_{regularized} = \underbrace{-\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(\hat{y}^{(i)}\right) + (1-y^{(i)})\log\left(1-\hat{y}^{(i)}\right)\right)}_{\text{cross-entropy cost}} + \underbrace{\frac{1}{m}\frac{\lambda}{2}\sum_{l}\sum_{k}\sum_{j} W_{k,j}^{[l]2}}_{\text{L2 regularization cost}}$$

$$\frac{\partial J}{\partial W^{[l]}} = dW^{[l]} = \{from\ backprop.\} + \frac{\lambda}{m}W^{[l]}$$

$$W^{[l]} = W^{[l]} - \eta\, dW^{[l]}$$

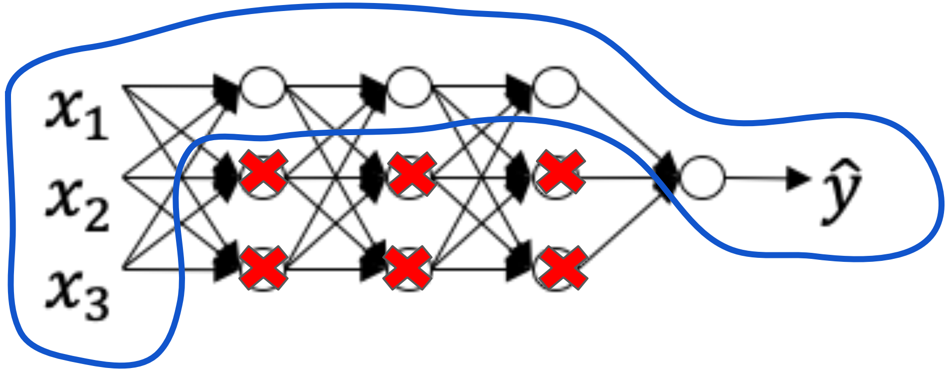
$$W^{[l]} = W^{[l]} - \eta\left[\{from\ backprop.\} + \frac{\lambda}{m}W^{[l]}\right]$$

"Weight Decay"

$$W^{[l]} = \boxed{(1 - \frac{\eta\lambda}{m})}W^{[l]} - \eta\left[\{from\ backprop.\}\right]$$

How does regularization prevent overfitting?

Intuition #01



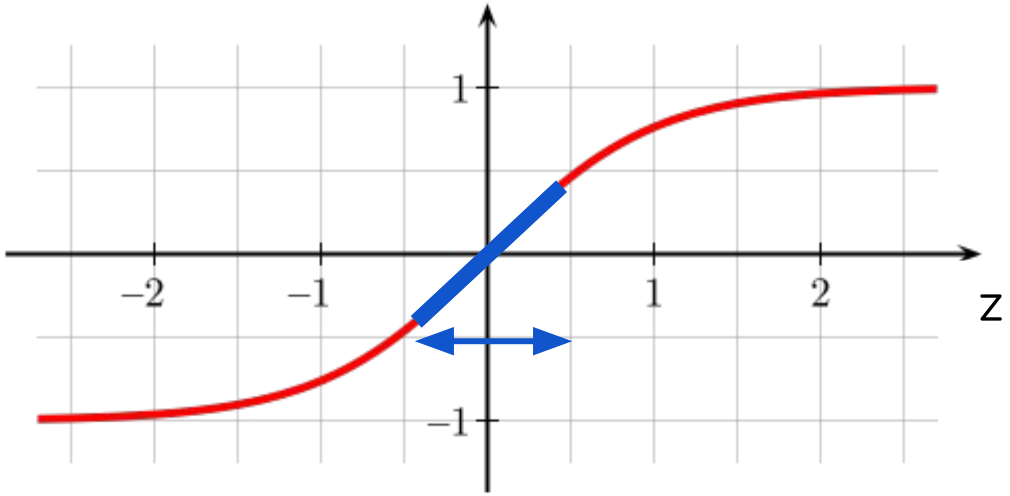
$$W^{[l]} = (1 - \frac{\eta\lambda}{m})W^{[l]} - \eta\,[\{from\ backprop.\}$$

$$\lambda \uparrow \quad \Rightarrow \quad W^{[l]} \approx 0$$

How does regularization prevent overfitting?

Intuition #02

Tanh (z)



$$\lambda \uparrow \quad \Rightarrow \quad W^{[l]} \approx 0$$

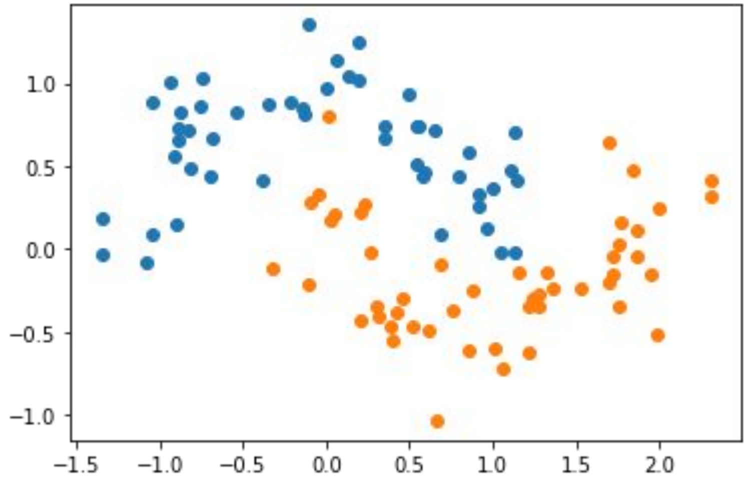
$$Z = \alpha^{[l-1]} W^{[l]} + b^{[l]}$$

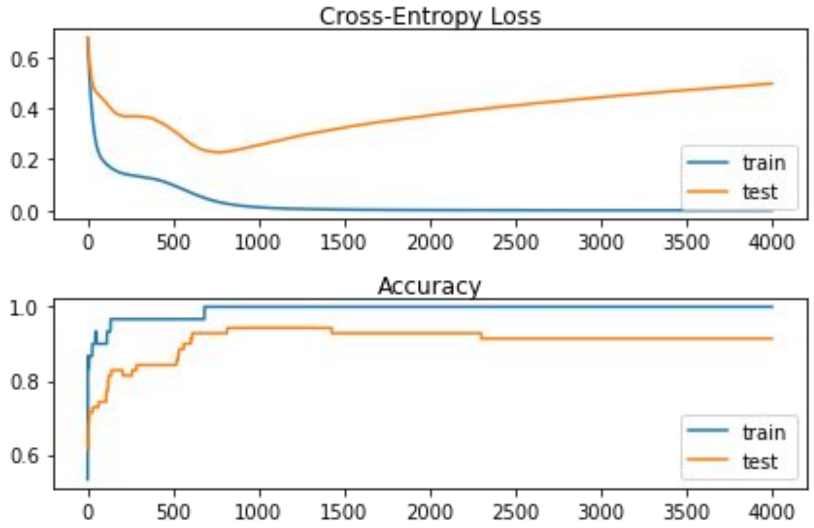
Weight Regularization Keras API

```
model = tf.keras.Sequential([
                             tf.keras.layers.Dense(500,activation=tf.nn.relu,
                                                   kernel_regularizer=tf.keras.regularizers.l2(l=0.01)),
                             tf.keras.layers.Dense(1, activation = tf.nn.sigmoid)
                             ])
```

$$J_{regularized} = \underbrace{-\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(\hat{y}^{(i)}\right) + (1-y^{(i)})\log\left(1-\hat{y}^{(i)}\right)\right)}_{\text{cross-entropy cost}} + \underbrace{\frac{1}{m}\frac{\lambda}{2}\sum_{l}\sum_{k}\sum_{j} W_{k,j}^{[l]2}}_{\text{L2 regularization cost}}$$

Weight Regularization Case Study

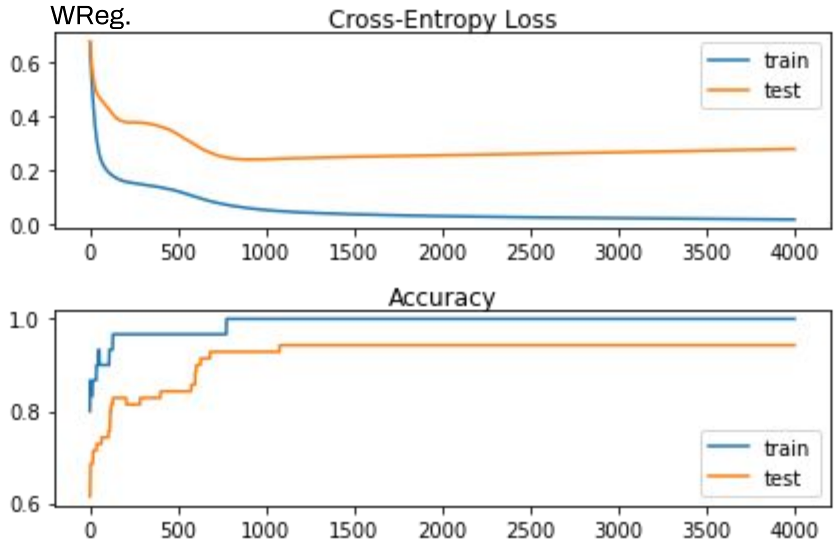


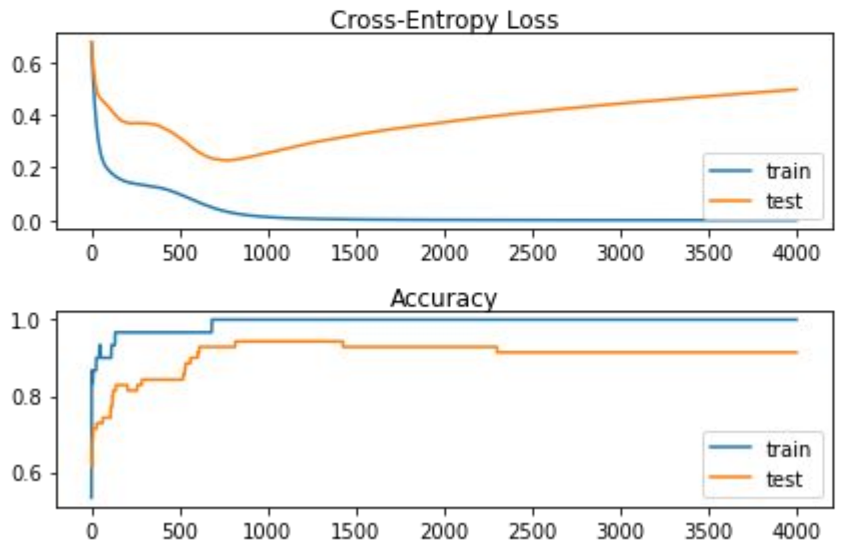


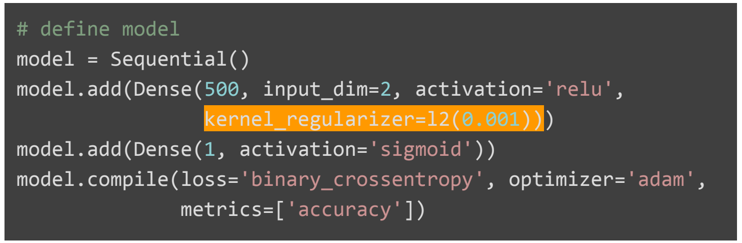
```
# define model
model = Sequential()
model.add(Dense(500, input_dim=2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy',optimizer='adam',
              metrics=['accuracy'])
```

Weight Regularization Case Study

WReg.



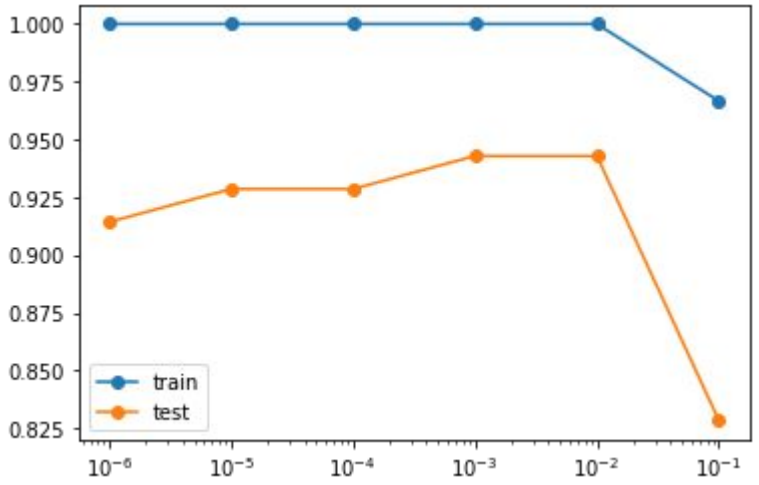




```
# define model
model = Sequential()
model.add(Dense(500, input_dim=2, activation='relu',
                kernel_regularizer=l2(0.001)))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam',
              metrics=['accuracy'])
```

Weight Regularization Case Study

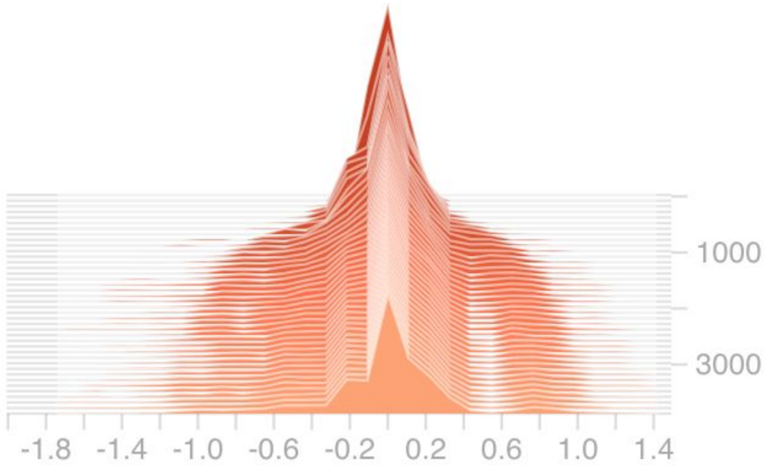
[Extensions]



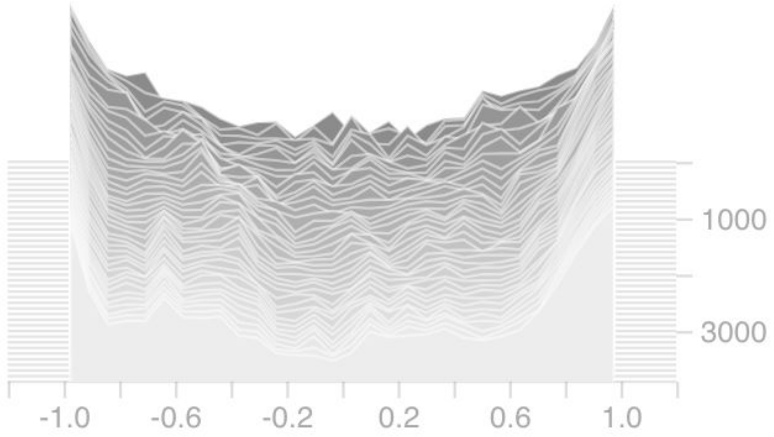
- **Try Alternates:** update the example to use L1 or the combined L1L2 methods instead of L2 regularization.
- **Report Weight Norm**: update the example to calculate the magnitude of the network weights and demonstrate that regularization indeed made the magnitude smaller.
- **Regularize Output Layer**: update the example to regularize the output layer of the model and compare the results.
- **Regularize Bias:** update the example to regularize the bias weight and compare the results.

Force small weights with weights constraints

Model without regularization



Model with weight constraints



https://www.tensorflow.org/tensorboard

Force small weights with weights constraints

```
filter = tf.keras.constraints.UnitNorm()

data = np.arange(3).reshape(3, 1).astype(np.float32)
print(data)

[[0.]
 [1.]
 [2.]]
```

```
filter(data)
<tf.Tensor: shape=(3, 1), dtype=float32, numpy=
array([[0.        ],
       [0.4472136],
       [0.8944272]], dtype=float32)>
```

```
np.linalg.norm(filter(data))
0.99999994
```

```
np.linalg.norm(data)
2.236068

data/np.linalg.norm(data)
array([[0.        ],
       [0.4472136],
       [0.8944272]], dtype=float32)
```

Force small weights with weights constraints

```
filter = tf.keras.constraints.UnitNorm()
data = np.arange(6).reshape(3, 2).astype(np.float32)
data
array([[0., 1.],
       [2., 3.],
       [4., 5.]], dtype=float32)
```

```
filter(data)
<tf.Tensor: shape=(3, 2), dtype=float32, numpy=
array([[0.        , 0.16903085],
       [0.4472136 , 0.50709254],
       [0.8944272 , 0.8451542 ]], dtype=float32)>
```

```
np.linalg.norm(data,axis=0)
array([4.472136, 5.91608 ], dtype=float32)

data/np.linalg.norm(data,axis=0)
array([[0.        , 0.16903085],
       [0.4472136 , 0.50709254],
       [0.8944272 , 0.8451542 ]], dtype=float32)
```

```
np.linalg.norm(filter(data),axis=0)
array([0.99999994, 0.99999994], dtype=float32)
```

`class MaxNorm` : MaxNorm weight constraint.

`class MinMaxNorm` : MinMaxNorm weight constraint.

`class NonNeg` : Constrains the weights to be non-negative.

`class RadialConstraint` : Constrains `Conv2D` kernel weights to be the same for each radius.

`class UnitNorm` : Constrains the weights incident to each hidden unit to have unit norm.

`class max_norm` : MaxNorm weight constraint.

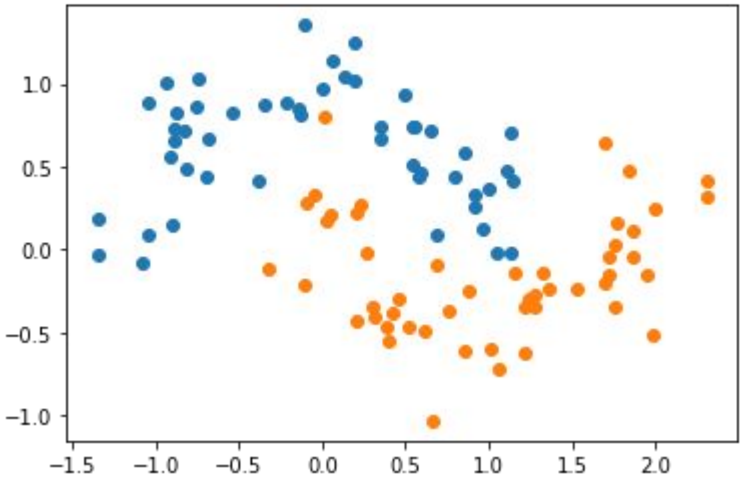
`class min_max_norm` : MinMaxNorm weight constraint.

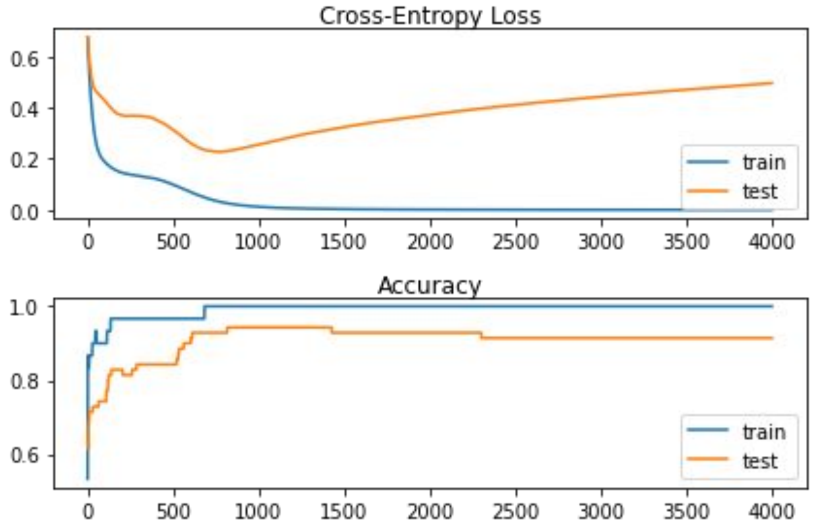
`class non_neg` : Constrains the weights to be non-negative.

`class radial_constraint` : Constrains `Conv2D` kernel weights to be the same for each radius.

`class unit_norm` : Constrains the weights incident to each hidden unit to have unit norm.

Weight Constraints Case Study

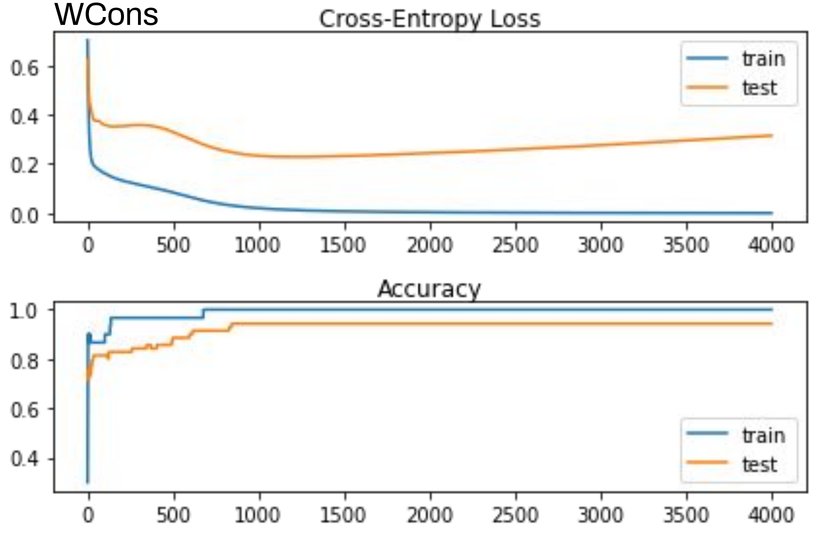


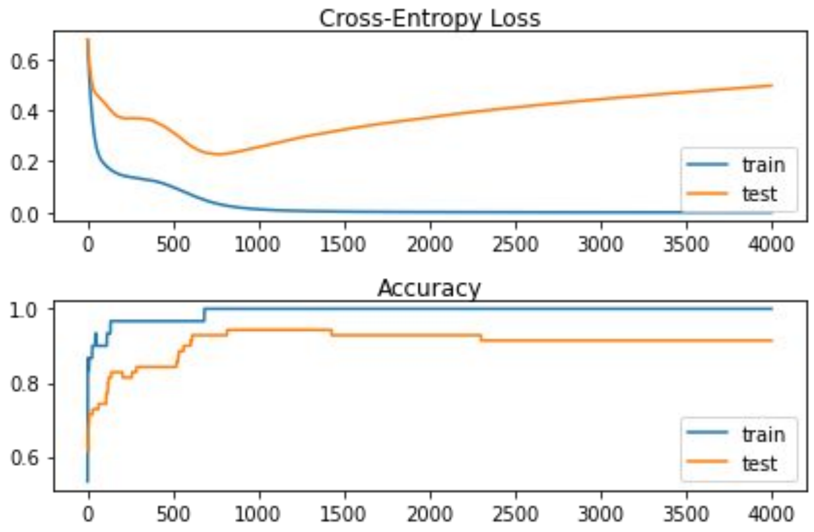


```
# define model
model = Sequential()
model.add(Dense(500, input_dim=2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy',optimizer='adam',
              metrics=['accuracy'])
```

Weight Constraints Case Study

WCons



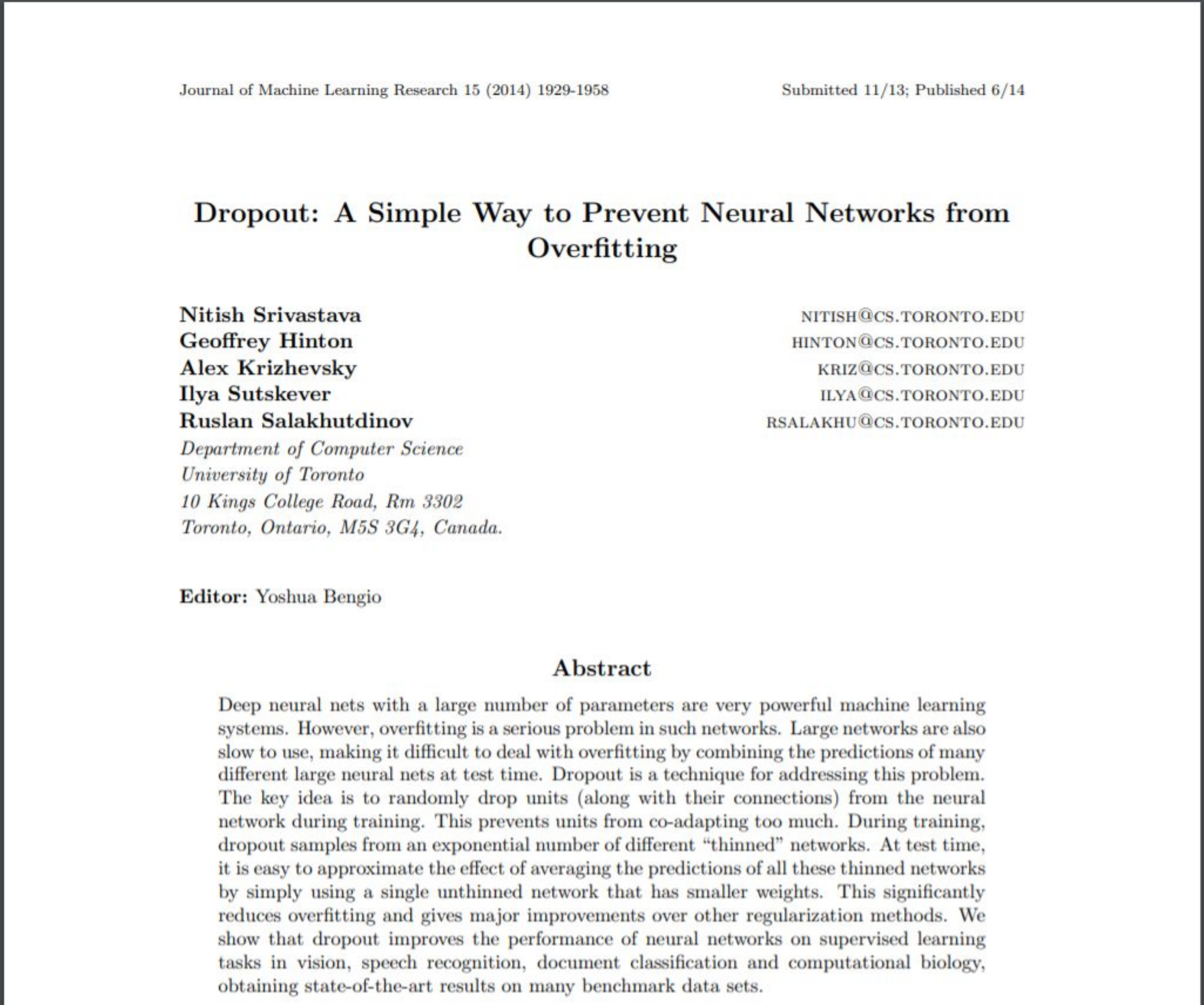


```
# define model
model = Sequential()
model.add(Dense(500, input_dim=2, activation='relu',
          kernel_constraint=unit_norm()))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam',
              metrics=['accuracy'])
```

Weight Constraints Case Study

[Extensions]

- **Weight constraints can improve generalization** when used in conjunction with other regularization methods, like L2 or Dropout.
- **Report Weight Norm**: update the example to calculate the magnitude of the unit weights and demonstrate that the constraint indeed made the magnitude smaller.
- **Constrain Output Layer**: update the example to add a constraint to the output layer of the model and compare the results.
- **Constrain Bias**: update the example to add a constraint to the bias weight and compare the results.



Journal of Machine Learning Research 15 (2014) 1929-1958 Submitted 11/13; Published 6/14

Dropout: A Simple Way to Prevent Neural Networks from Overfitting

Nitish Srivastava NITISH@CS.TORONTO.EDU
Geoffrey Hinton HINTON@CS.TORONTO.EDU
Alex Krizhevsky KRIZ@CS.TORONTO.EDU
Ilya Sutskever ILYA@CS.TORONTO.EDU
Ruslan Salakhutdinov RSALAKHU@CS.TORONTO.EDU
Department of Computer Science
University of Toronto
10 Kings College Road, Rm 3302
Toronto, Ontario, M5S 3G4, Canada.

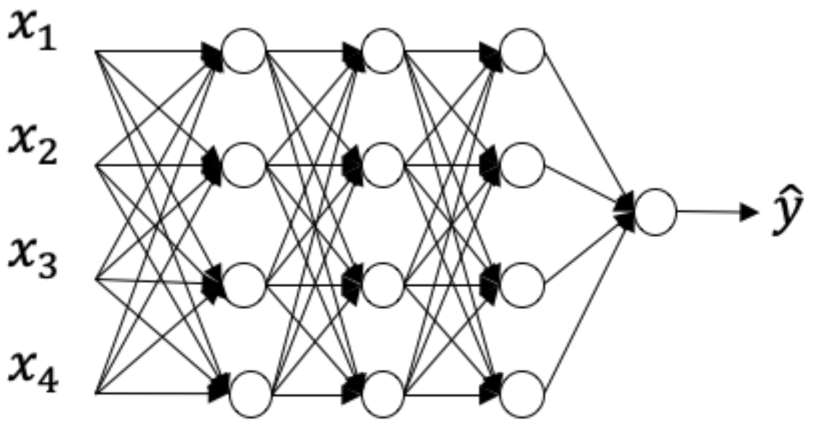
Editor: Yoshua Bengio

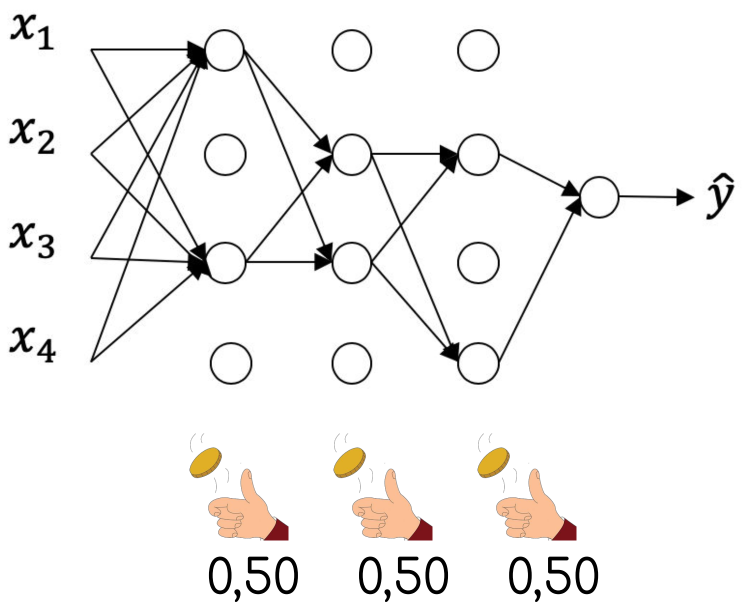
Abstract

Deep neural nets with a large number of parameters are very powerful machine learning systems. However, overfitting is a serious problem in such networks. Large networks are also slow to use, making it difficult to deal with overfitting by combining the predictions of many different large neural nets at test time. Dropout is a technique for addressing this problem. The key idea is to randomly drop units (along with their connections) from the neural network during training. This prevents units from co-adapting too much. During training, dropout samples from an exponential number of different "thinned" networks. At test time, it is easy to approximate the effect of averaging the predictions of all these thinned networks by simply using a single unthinned network that has smaller weights. This significantly reduces overfitting and gives major improvements over other regularization methods. We show that dropout improves the performance of neural networks on supervised learning tasks in vision, speech recognition, document classification and computational biology, obtaining state-of-the-art results on many benchmark data sets.

Dropout Regularization

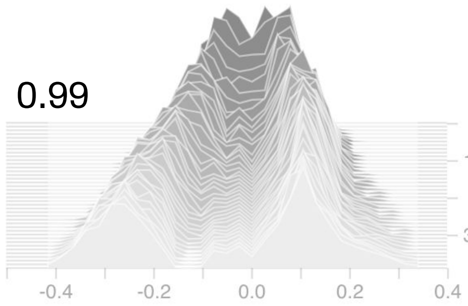
You implement Dropout regularization only while training the network. You do not apply it while running your testing data through it as you do not want any randomness in your predictions



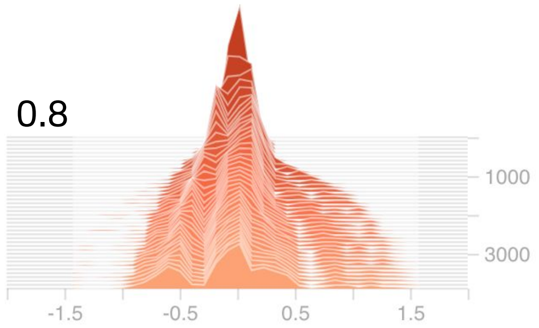


Dropout Regularization

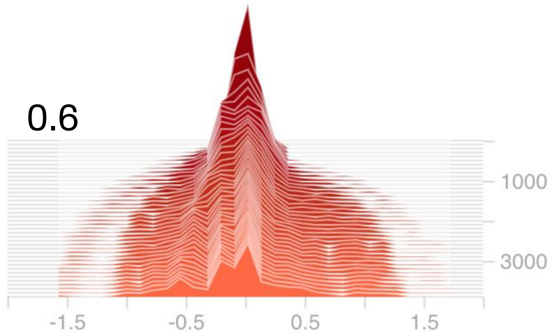
0.99



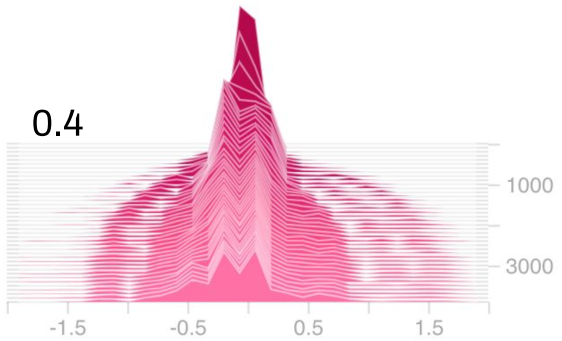
0.8



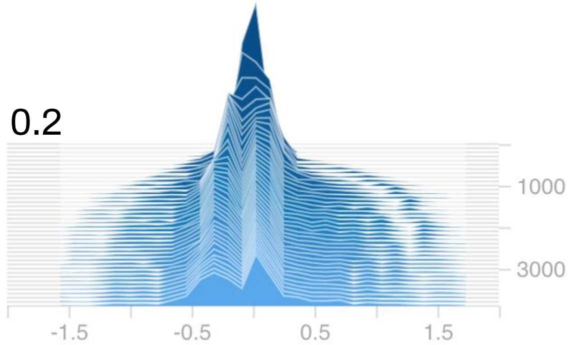
0.6



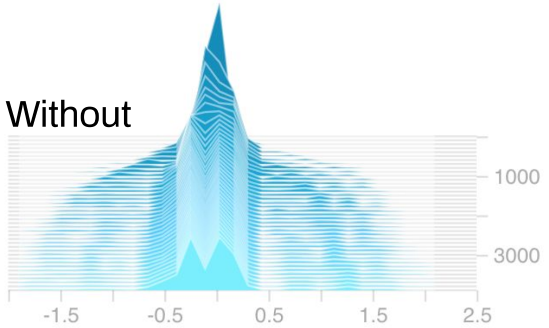
0.4



0.2



Without



Implementing Dropout ("Inverted Dropout")

Illustrate with layer "l" = 3

$$\begin{cases} keep\_prob & = 0.8 \\ 1 - keep\_prob & = 0.2 \end{cases}$$

$$d3 = np.randon.rand(a3.shape[0], a3.shape[1]) < keep\_prob$$

$$a3 = np.multiply(a3, d3)$$

$$a3/ = \underbrace{keep\_prob}$$

$$Z^{[4]} = \alpha^{[3]} W^{[4]} + b^{[4]}$$

It is necessary not to impact the value of Z.

Applying dropout for a input

keep_prob=0.8

```
import tensorflow as tf
import numpy as np

tf.random.set_seed(0)
layer = tf.keras.layers.Dropout(.2, input_shape=(2,))
data = np.arange(10).reshape(5, 2).astype(np.float32)
print(data)

[[0. 1.]
 [2. 3.]
 [4. 5.]
 [6. 7.]
 [8. 9.]]
```

```
outputs = layer(data, training=True)
print(outputs)

tf.Tensor(
[[ 0.    1.25]
 [ 2.5   3.75]
 [ 5.    6.25]
 [ 7.5   8.75]
 [10.    0.  ]], shape=(5, 2), dtype=float32)
```

output = output /keep_prob

Putting it all together

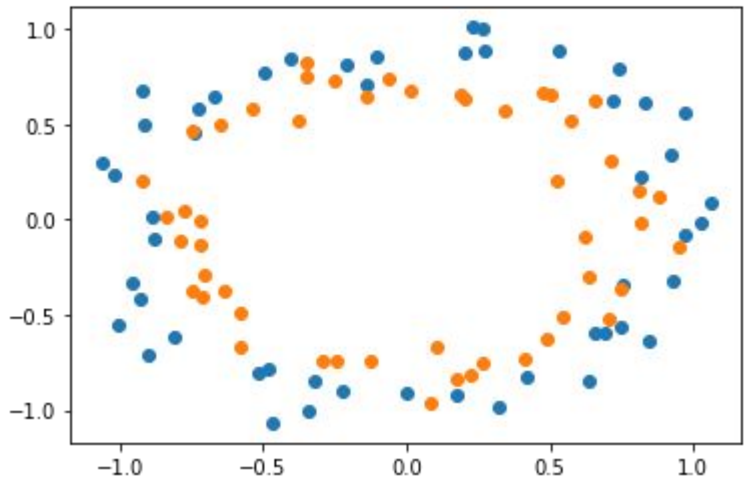
```
model_dropout = tf.keras.Sequential([
                             tf.keras.layers.Dense(20, activation=tf.nn.relu),
                             tf.keras.layers.Dropout(0.3),
                             tf.keras.layers.Dense(3, activation=tf.nn.relu),
                             tf.keras.layers.Dropout(0.3),
                             tf.keras.layers.Dense(1, activation = tf.nn.sigmoid)
                             ])
```

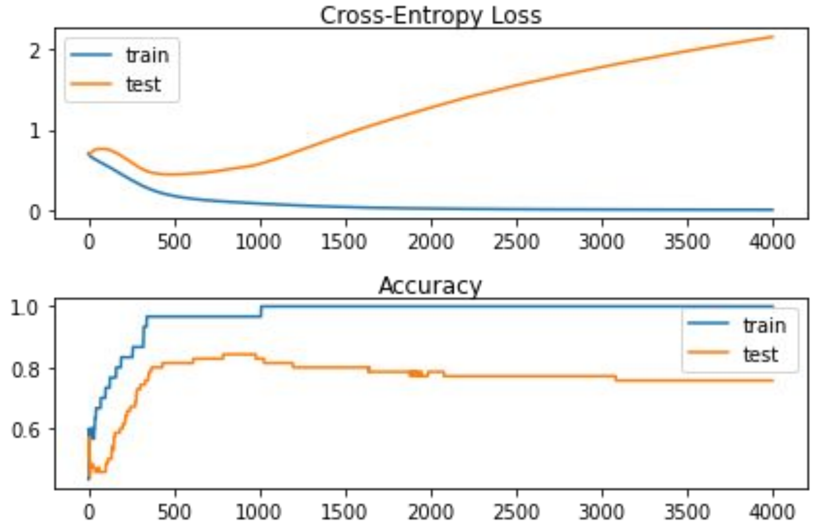
Rule of the thumb

>> You implement Dropout regularization only while training the network.

>> You do not apply it while running your testing data through it as you do not want any randomness in your predictions.

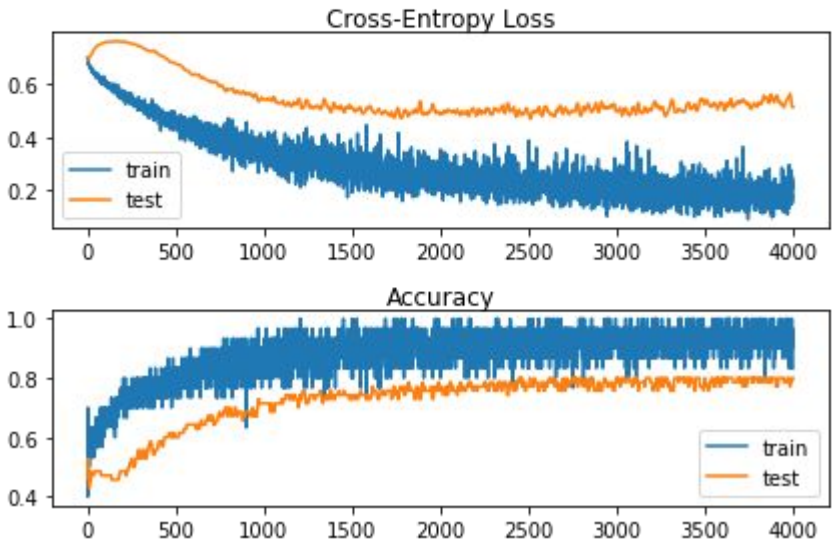
Dropout Case Study

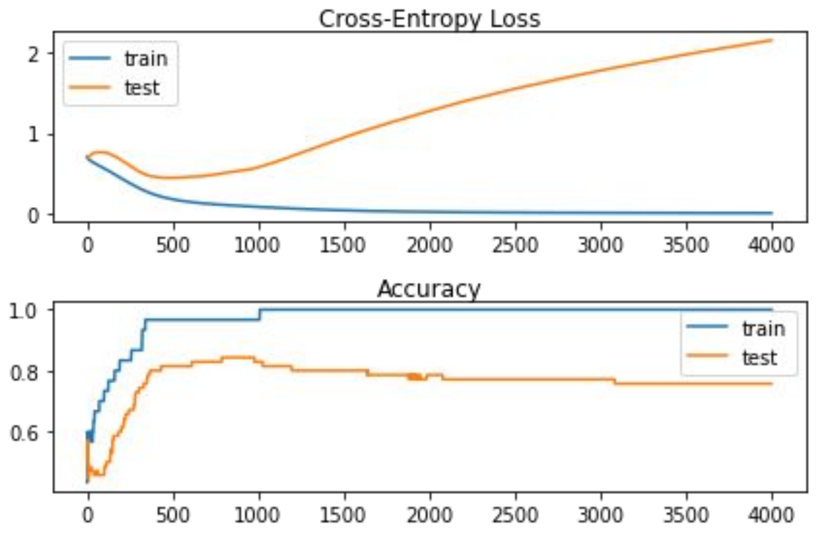




```
# define model
model = Sequential()
model.add(Dense(500, input_dim=2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy',optimizer='adam',
              metrics=['accuracy'])
```

Dropout Case Study





```
# define model
model = Sequential()
model.add(Dense(500, input_dim=2, activation='relu'))
model.add(Dropout(0.4))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam',
              metrics=['accuracy'])
```

Dropout Case Study [Extensions]

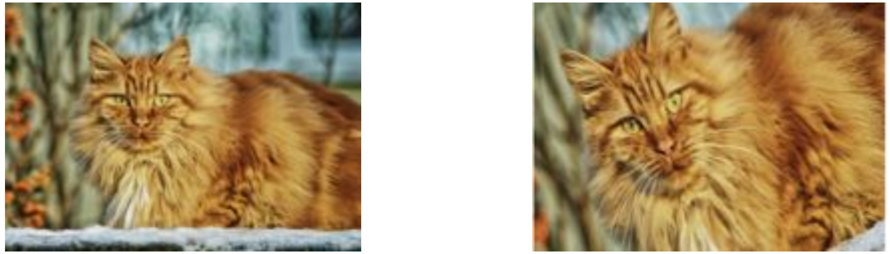
- **Input Dropout**: update the example to use dropout on the input variables and compare results.
- **Weight Constraint**: update the example to add a max-norm weight constraint to the hidden layer and compare results.
- **Repeated Evaluation**: update the example to repeat the evaluation of the overfit and dropout model and summarize and compare the average results.
- **Grid Search Rate**: develop a grid search of dropout probabilities and report the relationship between dropout rate and test set accuracy.

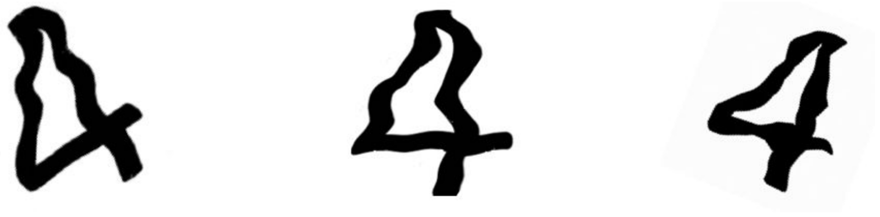
Promote Robustness with Noise

Other Regularization Methods

Original Data

Data Augmentation





How and Where to Add Noise

The most common type of noise used **during training** is the addition of **Gaussian noise** to input variables. Gaussian noise, or white noise, has a **mean of zero** and a **standard deviation of one**.

- **Add noise to activations**, i.e., the outputs of each layer.
- **Add noise to weights**, i.e., an alternative to the inputs.
- **Add noise to the gradients**, i.e., the direction to update weights.
- **Add noise to the outputs**, i.e., the labels or target variables.

Noise Regularization in Models

```
# import noise layer
from keras.layers import GaussianNoise
# define noise layer
layer = GaussianNoise(0.1)
```

Input

```
...
model.add(GaussianNoise(0.01, input_shape=(2,)))
...
```

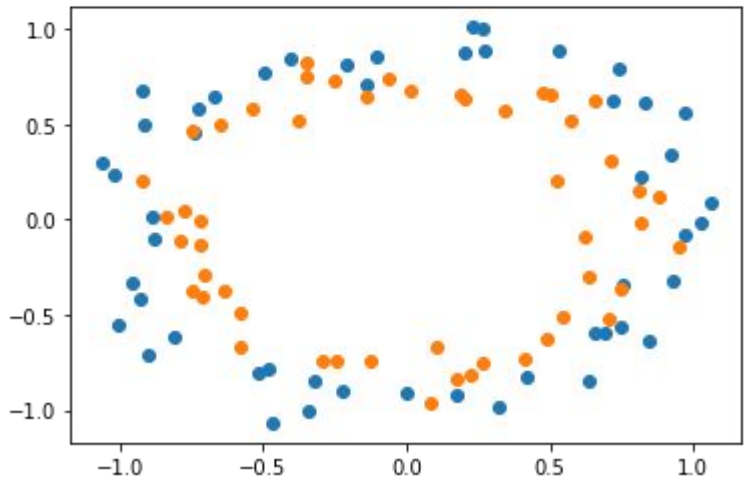
Before activation

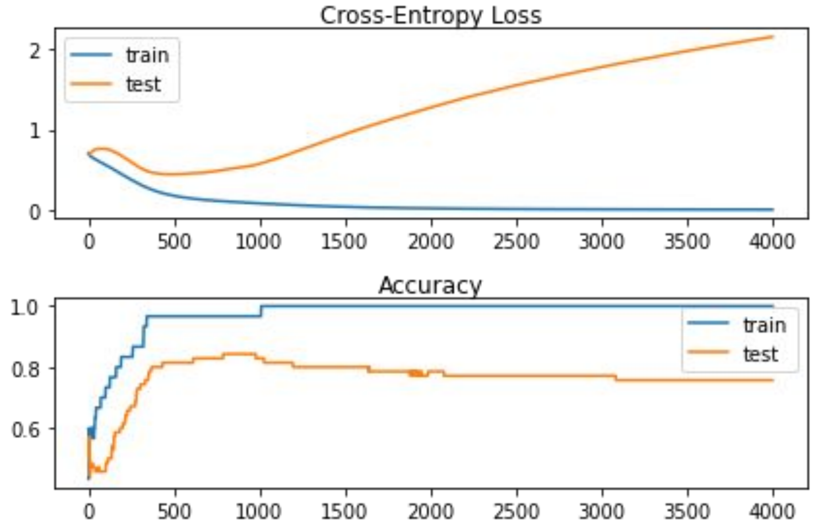
```
...
model.add(Dense(32))
model.add(GaussianNoise(0.1))
model.add(Activation('relu'))
model.add(Dense(32))
...
```

After activation

```
...
model.add(Dense(32, activation='reu'))
model.add(GaussianNoise(0.1))
model.add(Dense(32))
...
```

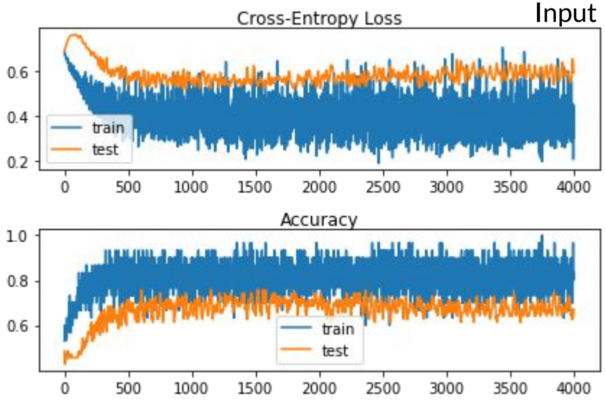
Noise Regularization Case Study



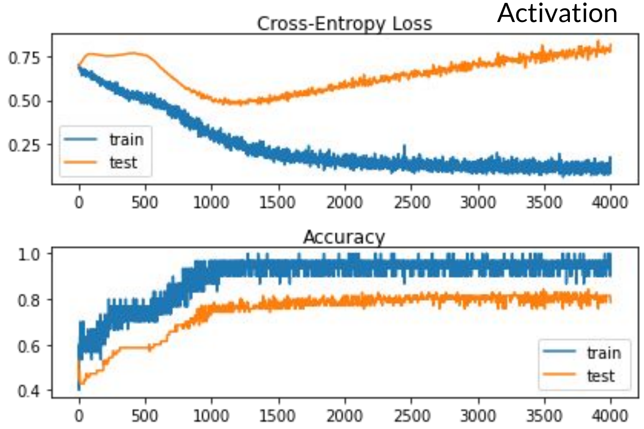


```
# define model
model = Sequential()
model.add(Dense(500, input_dim=2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy',optimizer='adam',
              metrics=['accuracy'])
```

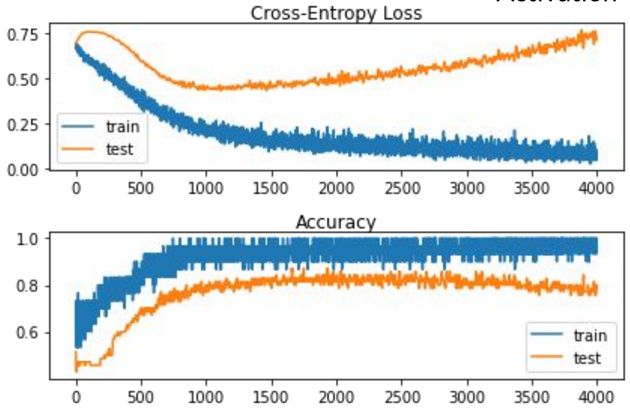
Input

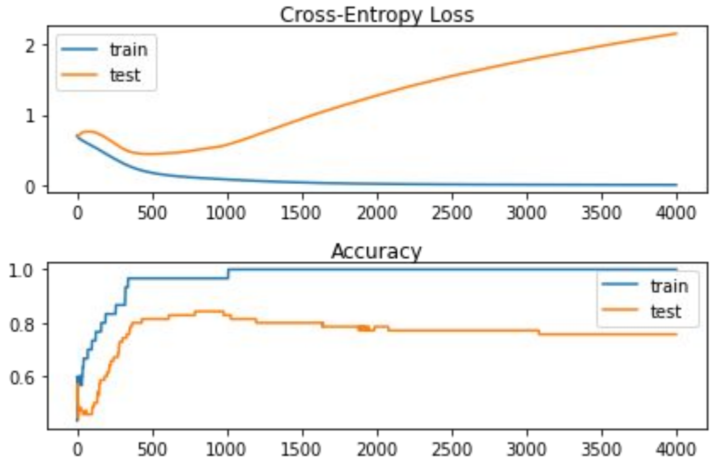


Before Activation



After Activation



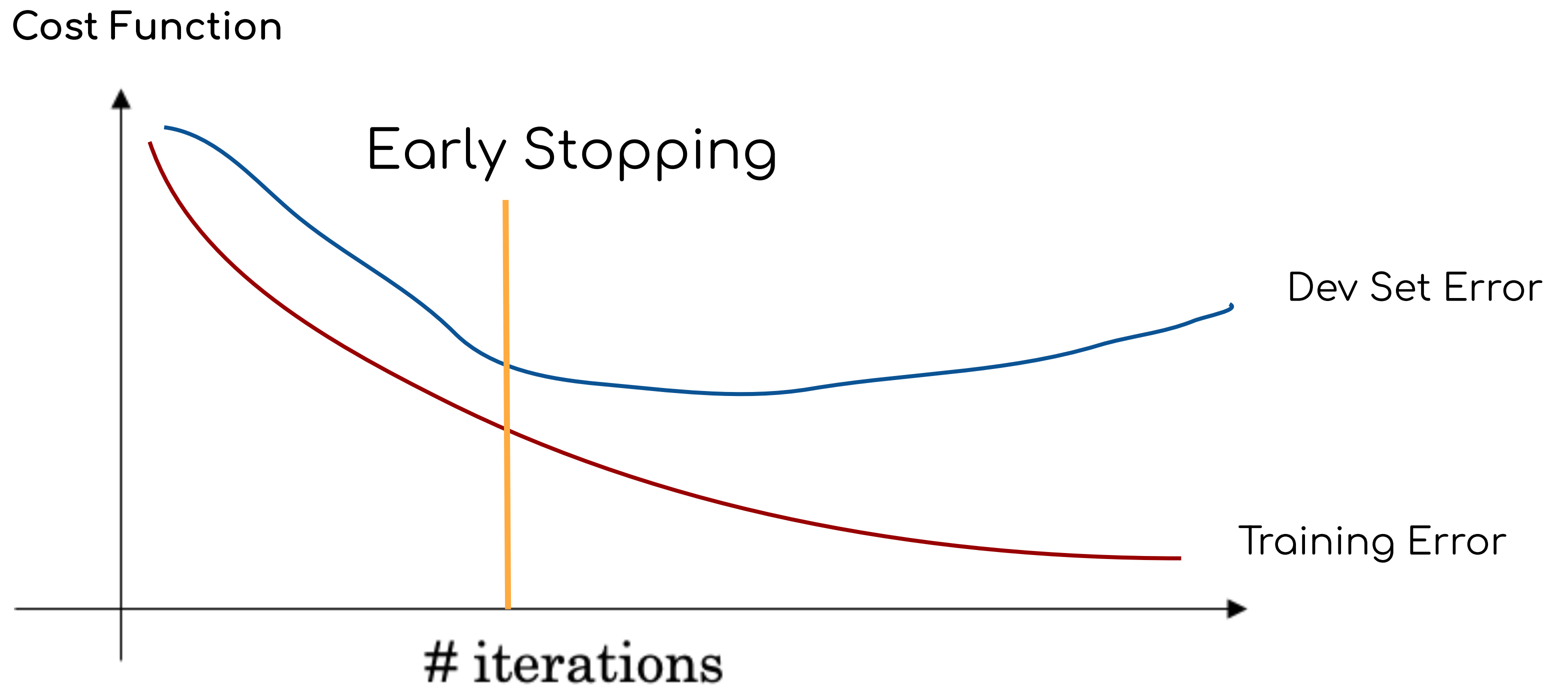


Noise Regularization Case Study

[Extensions]

- **Repeated Evaluation:** update the example to use repeated evaluation of the model with and without noise and report performance as the mean and standard deviation over repeats.
- **Grid Search Standard Deviation:** develop a grid search in order to discover the amount of noise that reliably results in the best performing model.
- **Input and Hidden Noise**: update the example to introduce noise at both the input and hidden layers of the model.

Halt training at the right time with Early Stopping



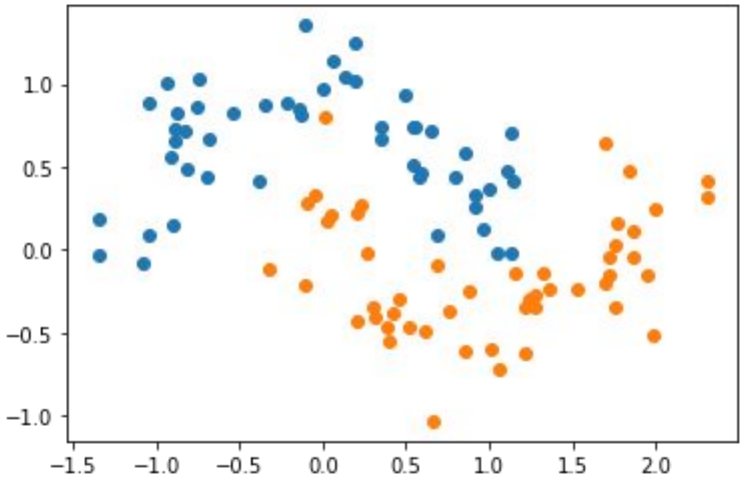
How to stop training early?

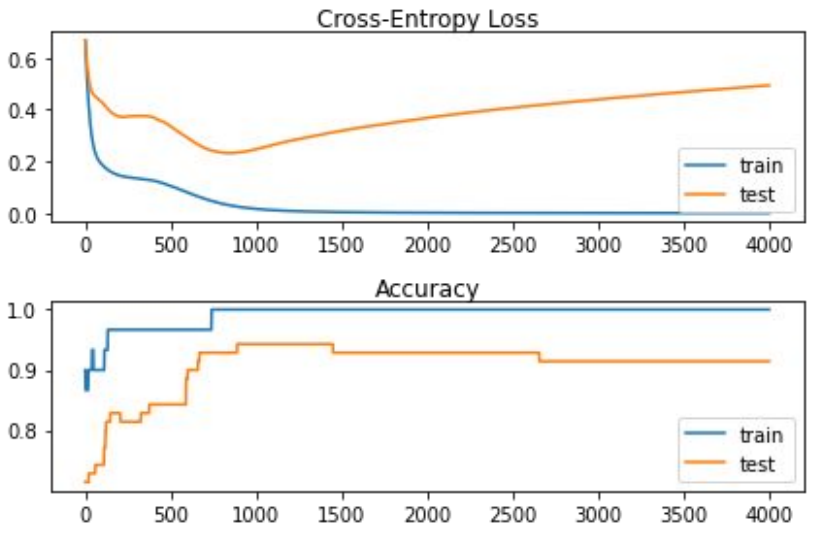
Early stopping requires that you configure your network to be **under constrained**, meaning that it has more capacity than is required for the problem.

There are three elements to using early stopping, they are:

- Monitoring model performance (frequency,loss).
- Trigger to stop training
    a. No change in metric over a given number of epochs.
    b. An absolute change in a metric.
    c. A decrease in performance observed over a given number of epochs.
    d. Average change in metric over a given number of epochs.
- The choice of model to use.

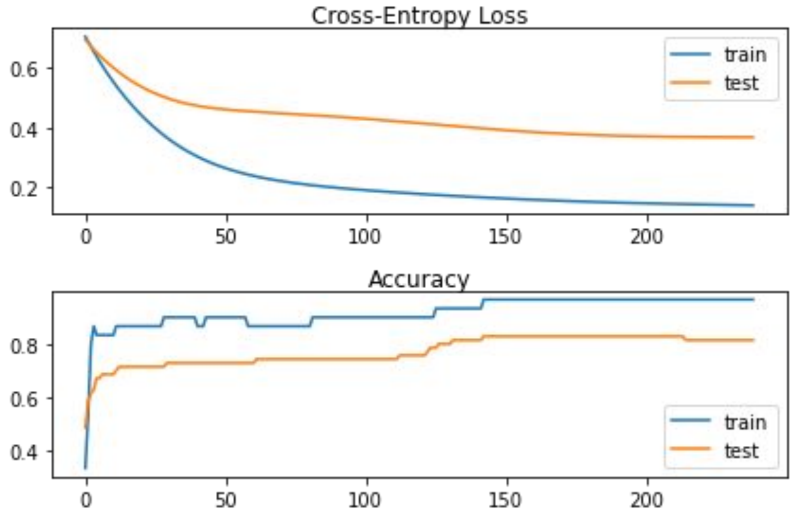
Early Stopping Case Study

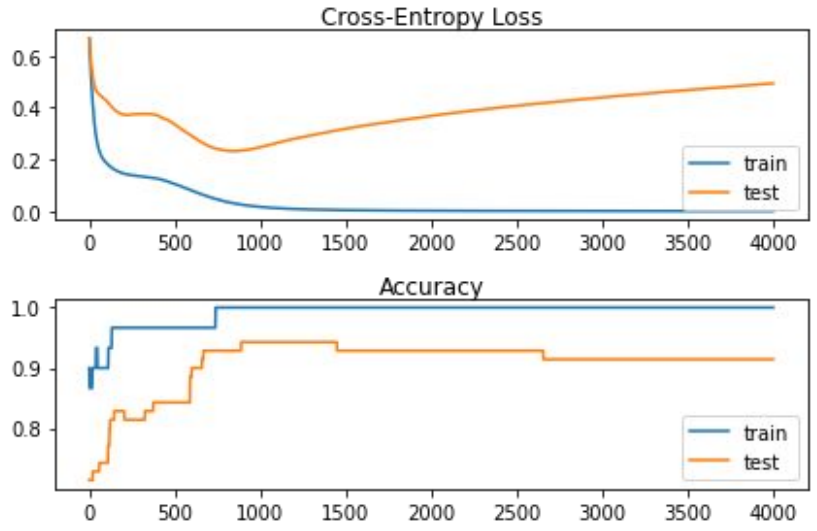




```
# define model
model = Sequential()
model.add(Dense(500, input_dim=2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy',optimizer='adam',
              metrics=['accuracy'])
```

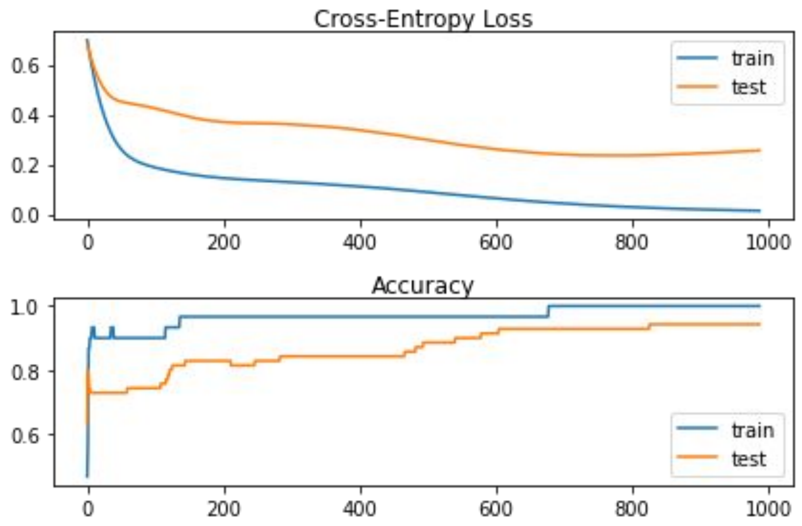
Early Stopping Case Study

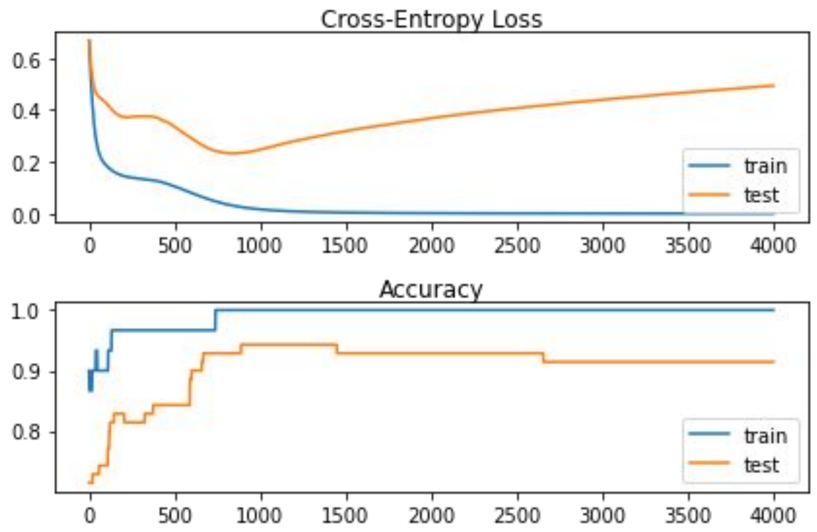




```
# simple early stopping
es = EarlyStopping(monitor='val_loss', mode='min', verbose=1)

# fit model
history = model.fit(train_x, train_y,
                    validation_data=(test_x, test_y),
                    epochs=4000, verbose=0, callbacks=[es])
```

Early Stopping Case Study





```
# patient early stopping
es = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=200)
# fit model
history = model.fit(train_x, train_y,
                    validation_data=(test_x, test_y),
                    epochs=4000, verbose=0, callbacks=[es])
```

Early Stopping Case Study

```
# simple early stopping
es = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=200)
mc = ModelCheckpoint('best_model.h5', monitor='val_accuracy',
                    mode='max', verbose=1, save_best_only=True)

# fit model
history = model.fit(train_x, train_y,
                   validation_data=(test_x, test_y),
                   epochs=4000, verbose=0, callbacks=[es, mc])

# load the saved model
saved_model = load_model('best_model.h5')
```

Always save the model weights if the model's performance on a holdout dataset is better than at the previous epoch

Early Stopping Case Study

[Extensions]

- **Use Accuracy**: update the example to monitor accuracy on the test dataset rather than loss, and plot learning curves showing accuracy.
- **Use True Validation Set**: update the example to split the training set into train and validation sets, then evaluate the model on the test dataset.
- **Regression Example**: create a new example of using early stopping to address overfitting on a simple regression problem and monitoring mean squared error.

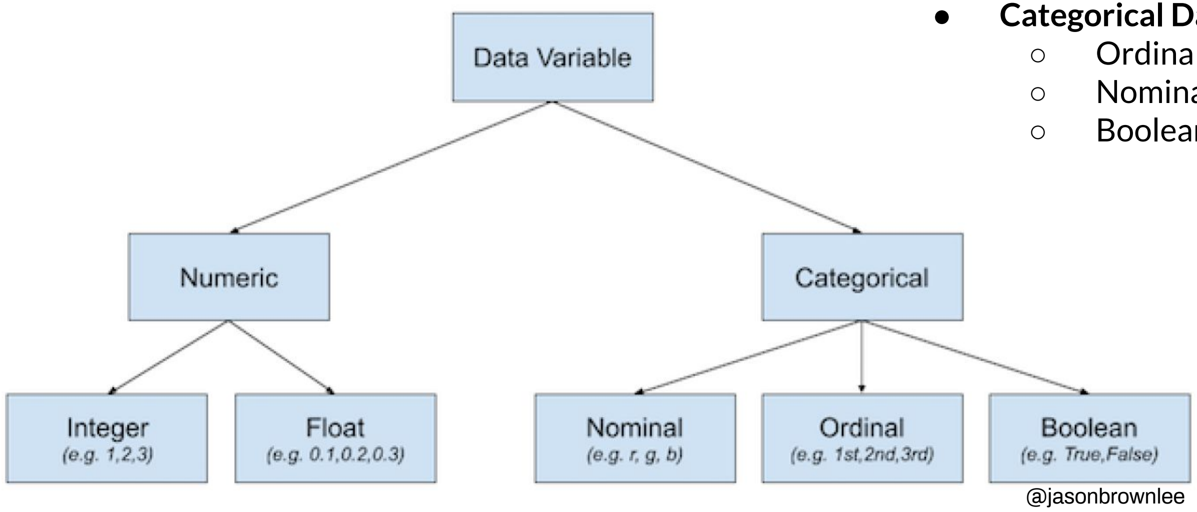
Neural Networks in Practice #02

Normalize Inputs, Vanishing/Exploding Gradients and Weight Initialization

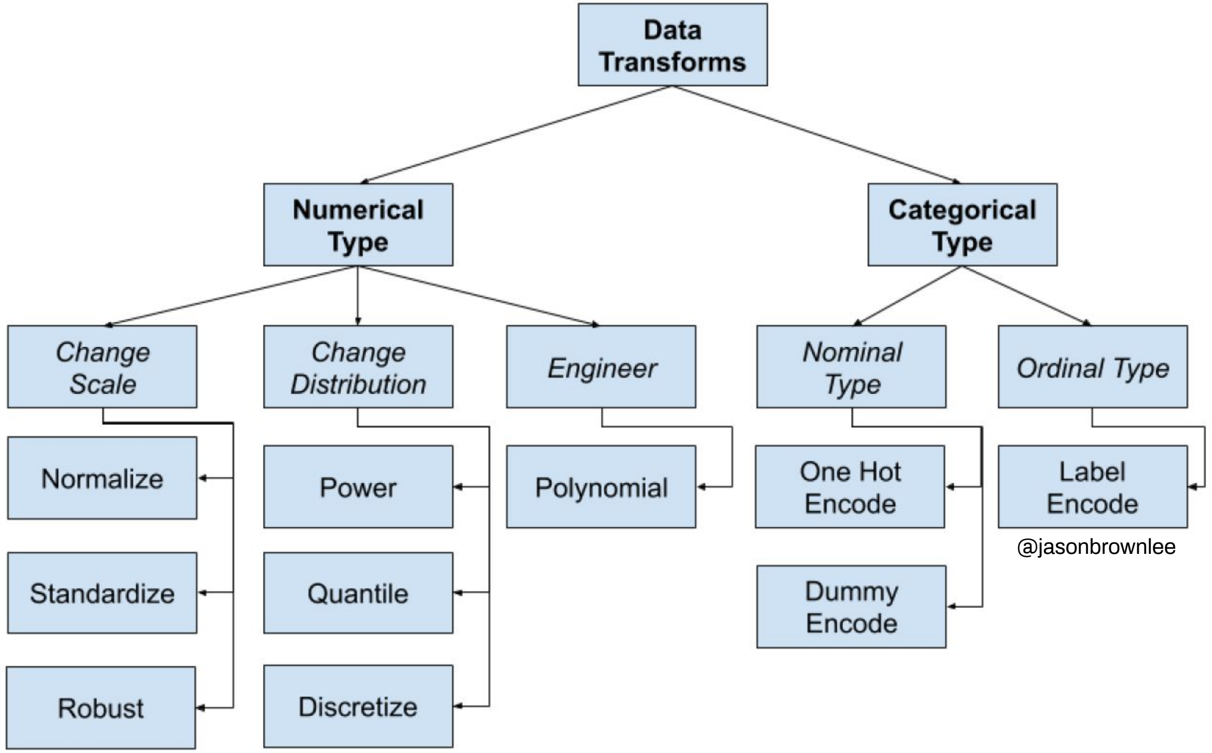
Data Transforms

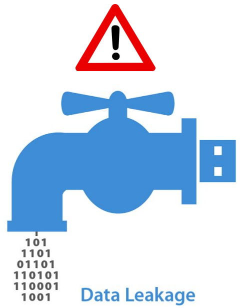
Data transforms are used to change the type or distribution of data variables.

- **Numeric Data Type**: number values.
  - Integer: Integers with no fractional part.
  - Float: Floating point values.
- **Categorical Data Type**: label values.
  - Ordinal: Labels with a rank ordering.
  - Nominal: Labels with no rank ordering.
  - Boolean: Values True and False.



Data Transforms





Data Leakage

```
# normalize the dataset
scaler = MinMaxScaler()
x = scaler.fit_transform(x)

# split into train and test sets
x_train, x_test, y_train, y_test = train_test_split(x,
                                                    y,
                                                    test_size=0.3,
                                                    random_state=1)
```

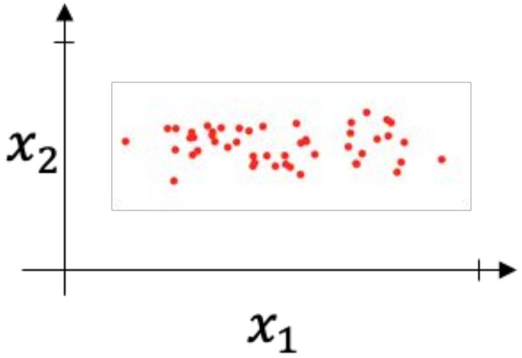
Wrong Way!!!!

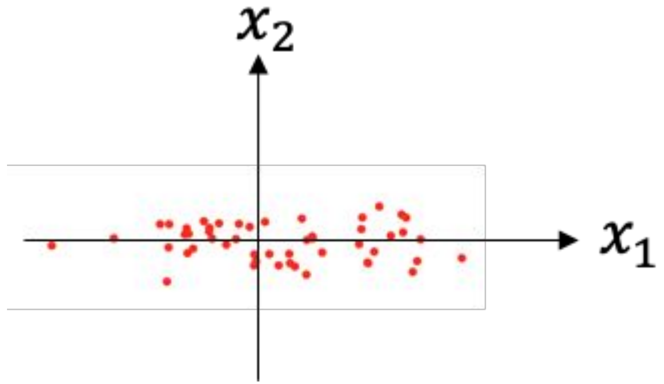
```
# split into train and test sets
x_train, x_test, y_train, y_test = train_test_split(x,
                                                    y,
                                                    test_size=0.3,
                                                    random_state=1)

# normalize the train dataset
scaler = MinMaxScaler()
x_train = scaler.fit_transform(x_train)

# scale the test dataset
x_test = scaler.transform(x_test)
```

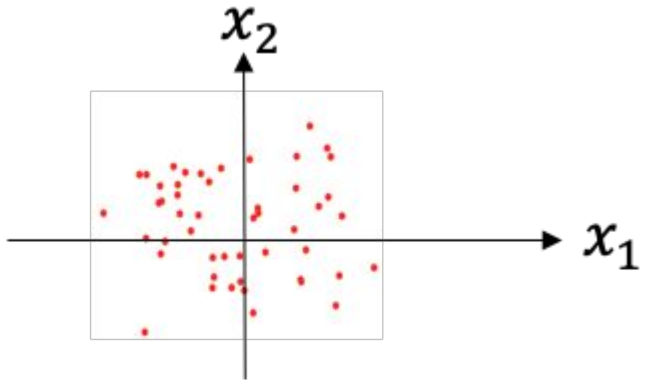
Right Way!!!!

Standardize Training Sets

Problem: x_1 and x_2 *have different scales*





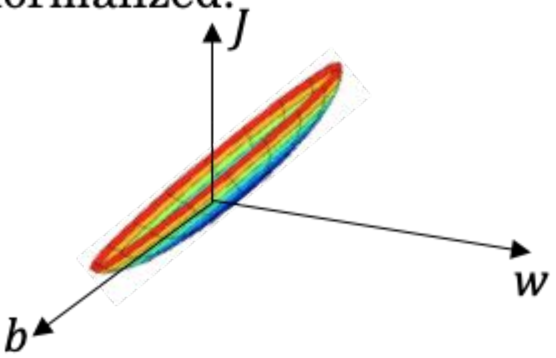


$$x - \mu$$

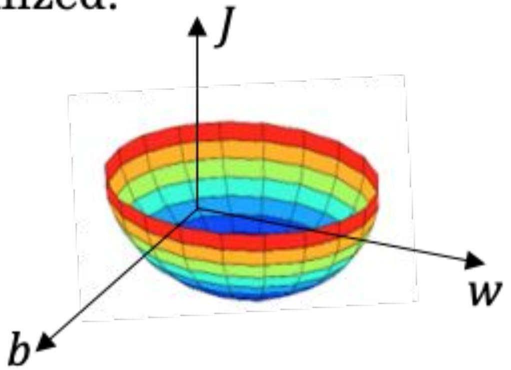
$$z\_score = \frac{x - \mu}{\sigma}$$

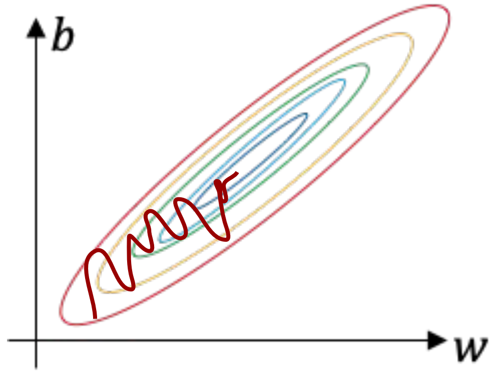
Why normalize inputs? Outputs?

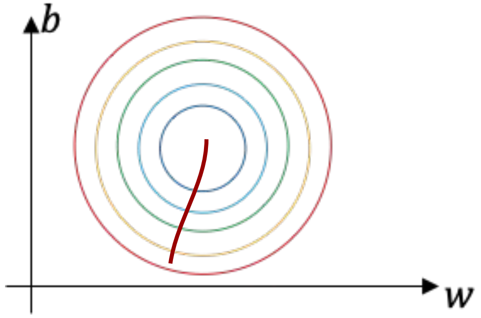
Unnormalized:



Normalized:

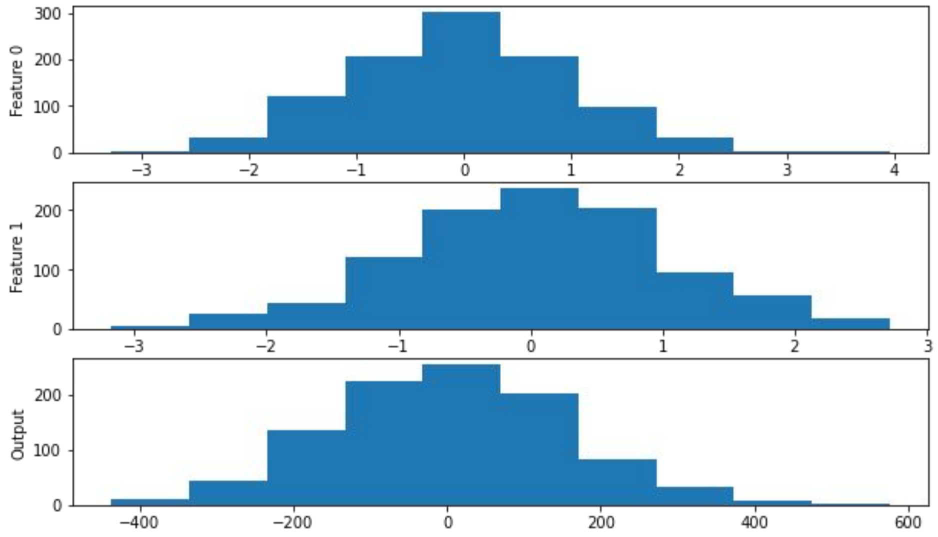




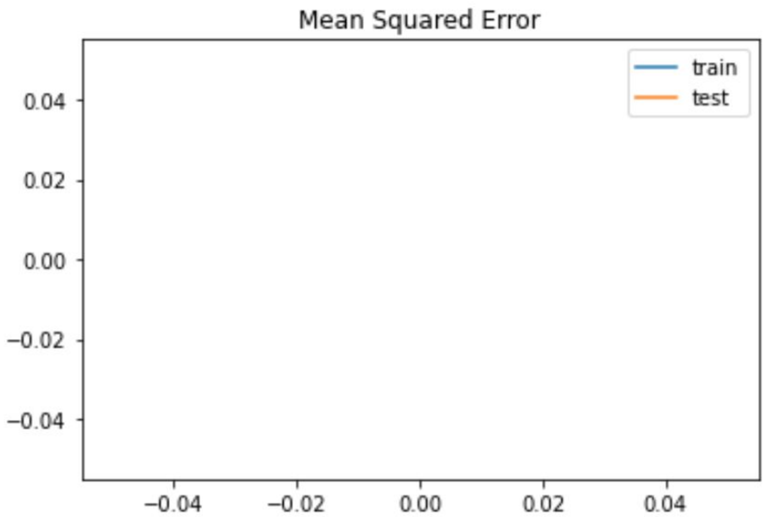


Data Scaling Case Study

MLP with unscaled data



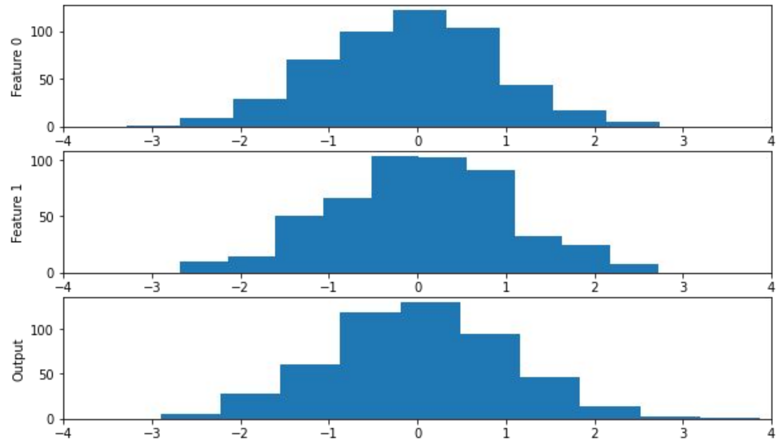
Train: nan, Test: nan

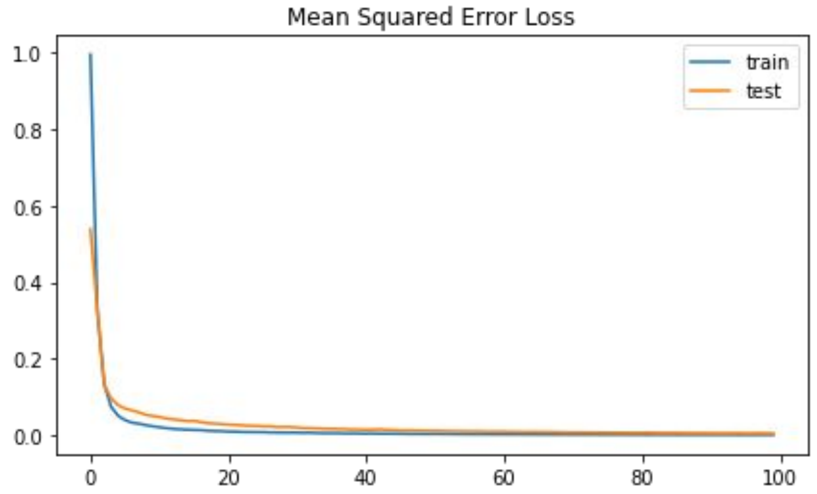


```
# define model
model = Sequential()
model.add(Dense(25, input_dim=20, activation='relu',
               kernel_initializer='he_uniform'))
model.add(Dense(1, activation='linear'))
```

Data Scaling Case Study

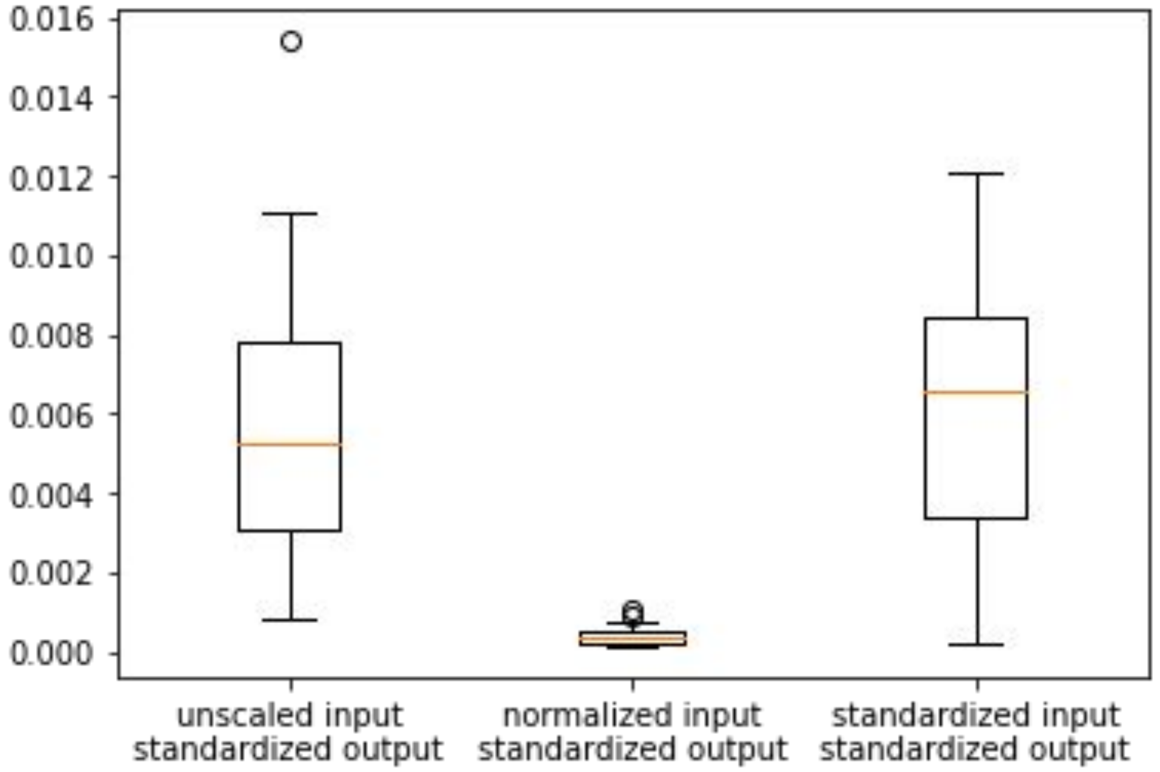
MLP with scaled output variables





```
# created scaler
scaler = StandardScaler()
# fit scaler on training dataset
scaler.fit(train_y)
# transform training dataset
train_y = scaler.transform(train_y)
# transform test dataset
test_y = scaler.transform(test_y)
```

Data Scaling Case Study

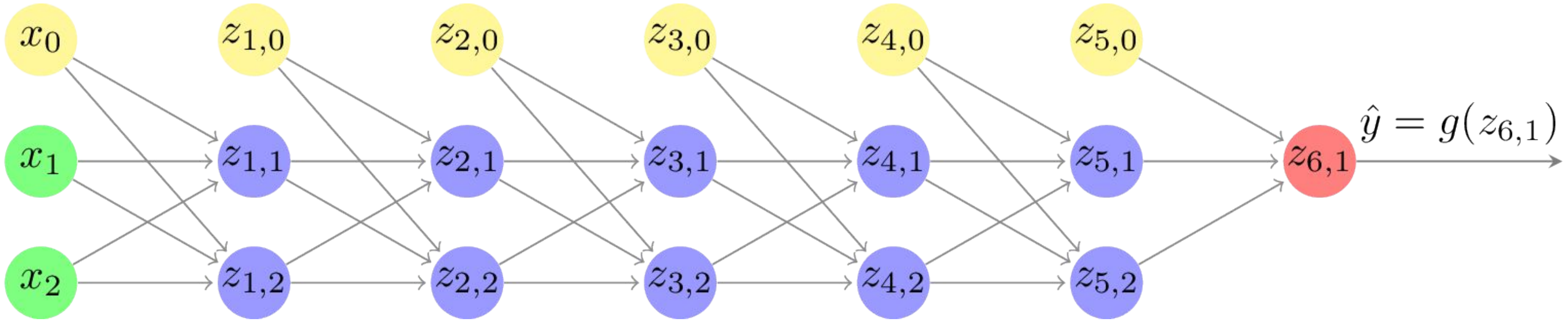


Data Scaling Case Study

[Extensions]

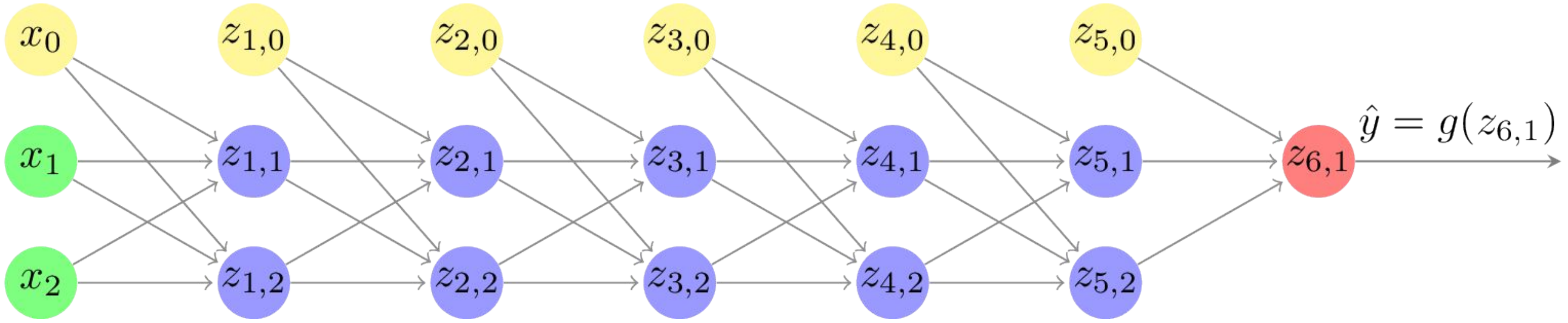
- **Normalize Target Variable**: update the example and normalize instead of standardize the target variable and compare results.
- **Compared Scaling for Target Variable**: update the example to compare standardizing and normalizing the target variable using repeated experiments and compare the results.
- **Other Scales**: update the example to evaluate other min/max scales when normalizing and compare performance, e.g [-1,1], and [0.0, 0.5]

Vanishing/Exploding Gradients



At every iteration of the optimization loop (forward, cost, backward, update), we observe that back propagated gradients are either amplified or minimized as you move from the output layer towards the input layer.

Vanishing/Exploding Gradients

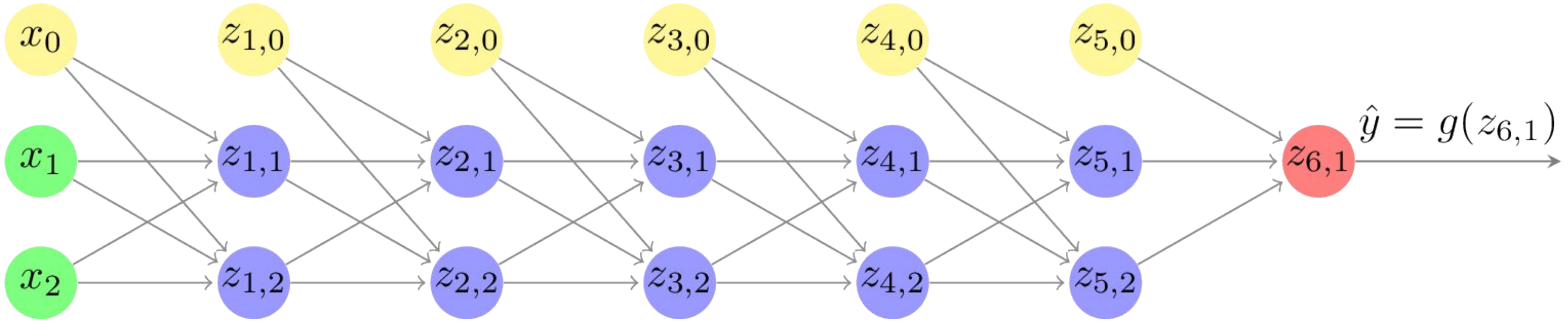


Assumptions

$$g^{[l]}(Z^{[l]}) = Z^{[l]}$$
$$b^{[l]} = 0$$
$$W^{[1]} = W^{[2]} = \ldots = W^{[L-1]} = W$$

$$Z^{[1]} = xW^{[1]}$$
$$a^{[1]} = g^{[1]}(Z^{[1]}) = Z^{[1]}$$
$$Z^{[2]} = a^{[1]}W^{[2]}$$
$$a^{[2]} = g^{[2]}(Z^{[2]}) = Z^{[2]} = xW^{[1]}W^{[2]}$$
$$\vdots$$
$$\hat{y} = a^{[L]} = xW^{l-1}W^{[L]}$$

Vanishing/Exploding Gradients

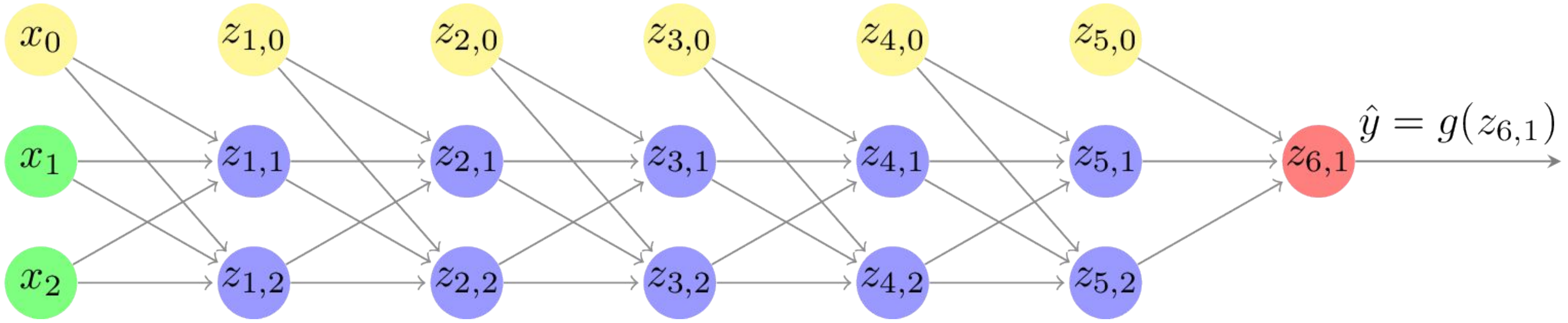


Case 1: A too-large initialization leads to exploding gradients

$$W^{[1]} = W^{[2]} = \cdots = W^{[L-1]} = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$$

$$Z^{[1]} = xW^{[1]}$$
$$a^{[1]} = g^{[1]}(Z^{[1]}) = Z^{[1]}$$
$$Z^{[2]} = a^{[1]}W^{[2]}$$
$$a^{[2]} = g^{[2]}(Z^{[2]}) = Z^{[2]} = xW^{[1]}W^{[2]}$$
$$\vdots$$
$$\hat{y} = a^{[L]} = xW^{l-1}W^{[L]}$$

Vanishing/Exploding Gradients



Case 2: A too-small initialization leads to vanishing gradients

$$W^{[1]} = W^{[2]} = \cdots = W^{[L-1]} = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}$$

$$Z^{[1]} = xW^{[1]}$$
$$a^{[1]} = g^{[1]}(Z^{[1]}) = Z^{[1]}$$
$$Z^{[2]} = a^{[1]}W^{[2]}$$
$$a^{[2]} = g^{[2]}(Z^{[2]}) = Z^{[2]} = xW^{[1]}W^{[2]}$$
$$\vdots$$
$$\hat{y} = a^{[L]} = xW^{l-1}W^{[L]}$$

How to find appropriate initialization values?

Weight Initialization

To prevent the gradients of the network's activations from vanishing or exploding, we will stick to the following rules of thumb:

1. The mean of the activations should be zero.
2. The variance of the activations should stay the same across every layer.

$$W^{[l]} = np.random.rand(n^{[l]}, n^{[l-1]}) \times factor$$

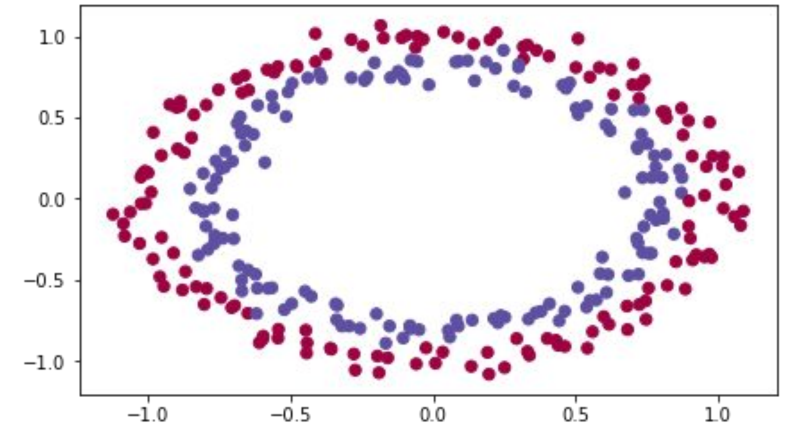
How to find appropriate initialization values?

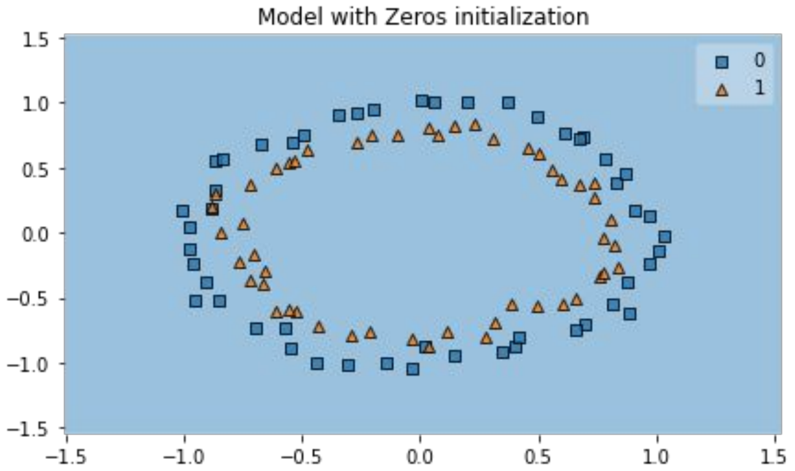
Weight Initialization

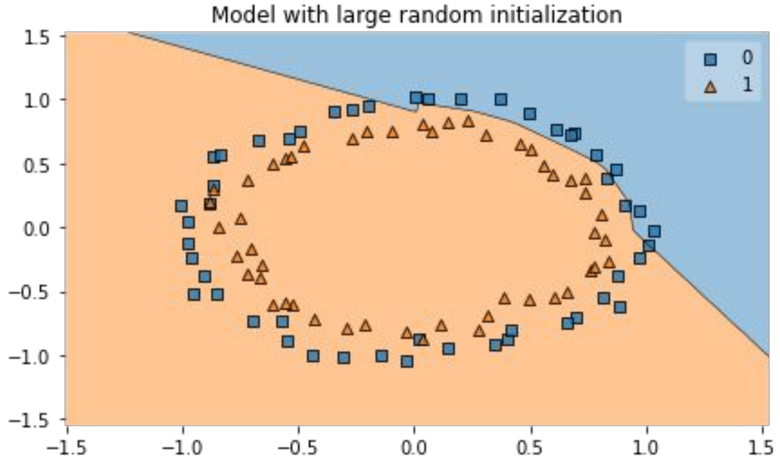
$$W^{[l]} = np.random.rand(n^{[l]}, n^{[l-1]}) \times factor$$

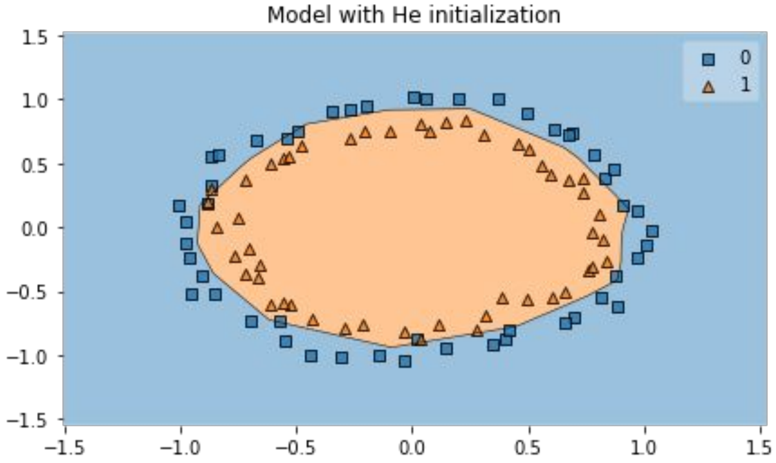
Activation Func.	Relu	Tanh
Factor	$\sqrt{\frac{2}{n^{[l-1]}}}$	$\sqrt{\frac{1}{n^{[l-1]}}}$ or $\sqrt{\frac{2}{n^{[l-1]} + n^{[l]}}}$
Author	He at al.	Xavier et al. or Yoshua Bengio et al.

https://www.deeplearning.ai/ai-notes/initialization/

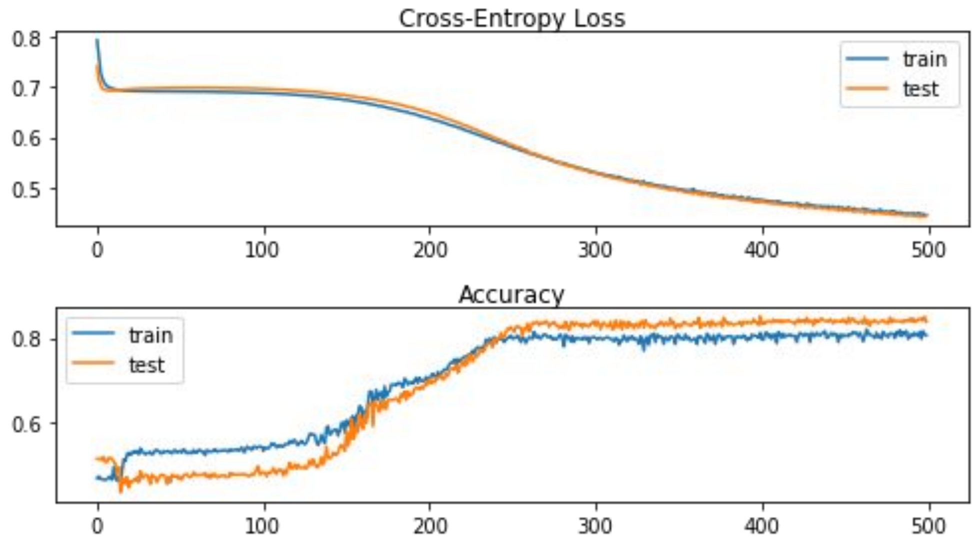

1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0

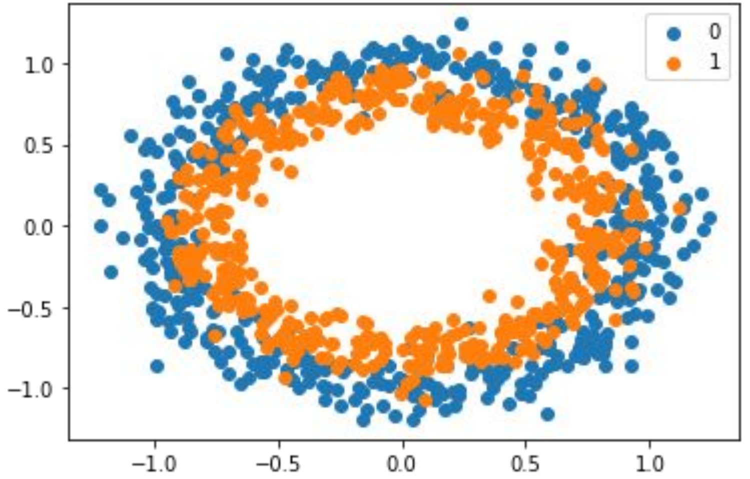

Model with Zeros initialization
0
1


Model with large random initialization
0
1


Model with He initialization
0
1

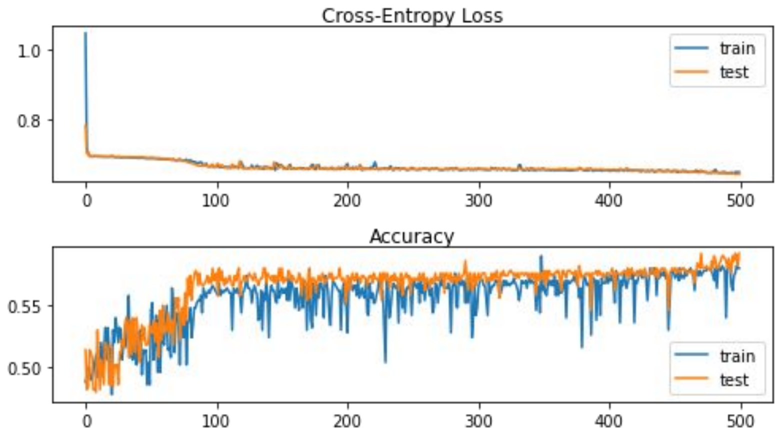
Fix Vanishing Gradients with Relu Case Study



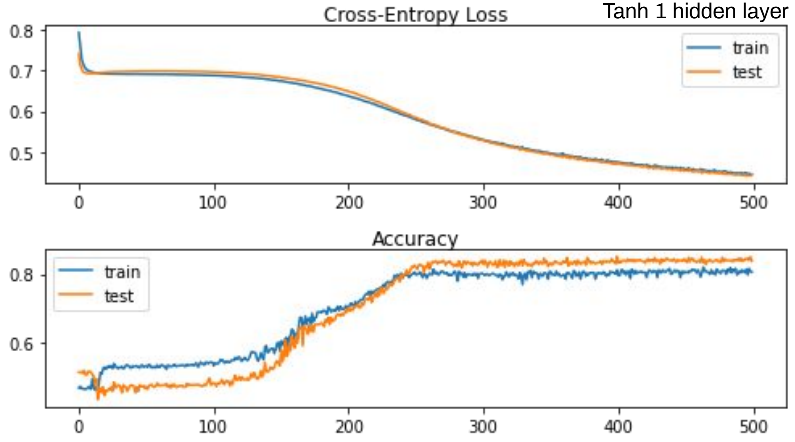


```
# It was also good practice to initialize the network weights
# to small random values from a uniform distribution.
init = RandomUniform(minval=0, maxval=1)
model.add(Dense(5, input_dim=2, activation='tanh', kernel_initializer=init))
model.add(Dense(1, activation='sigmoid', kernel_initializer=init))
```

Fix Vanishing Gradients with Relu Case Study

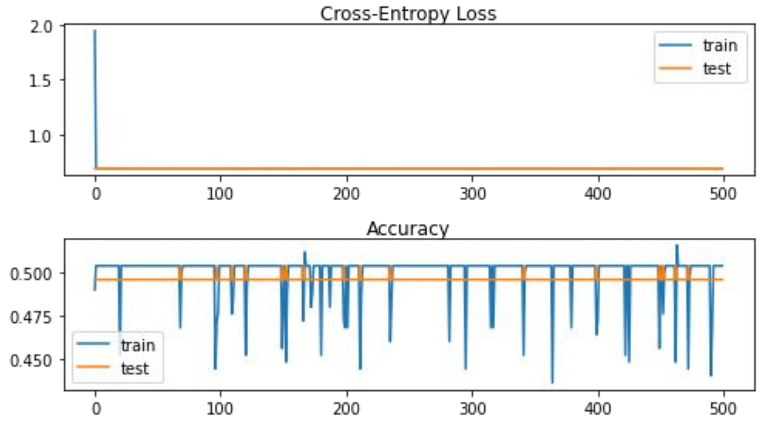


Tanh 1 hidden layer

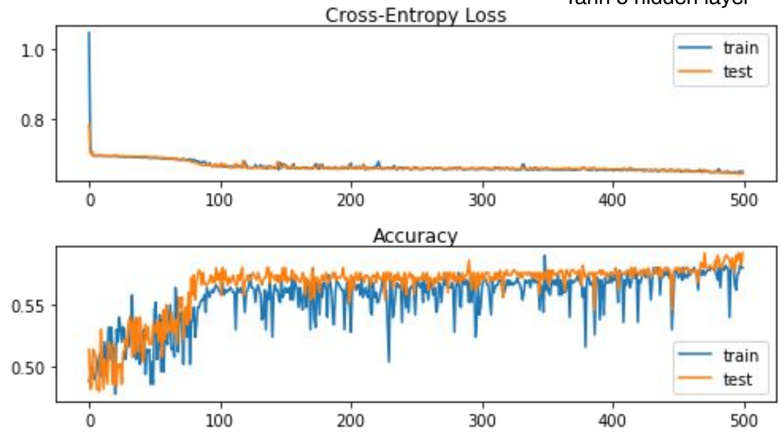


```
# define model
init = RandomUniform(minval=0, maxval=1)
model = Sequential()
model.add(Dense(5, input_dim=2, activation='tanh', kernel_initializer=init))
model.add(Dense(5, activation='tanh', kernel_initializer=init))
model.add(Dense(5, activation='tanh', kernel_initializer=init))
model.add(Dense(5, activation='tanh', kernel_initializer=init))
model.add(Dense(5, activation='tanh', kernel_initializer=init))
model.add(Dense(1, activation='sigmoid', kernel_initializer=init))
```

Fix Vanishing Gradients with Relu Case Study

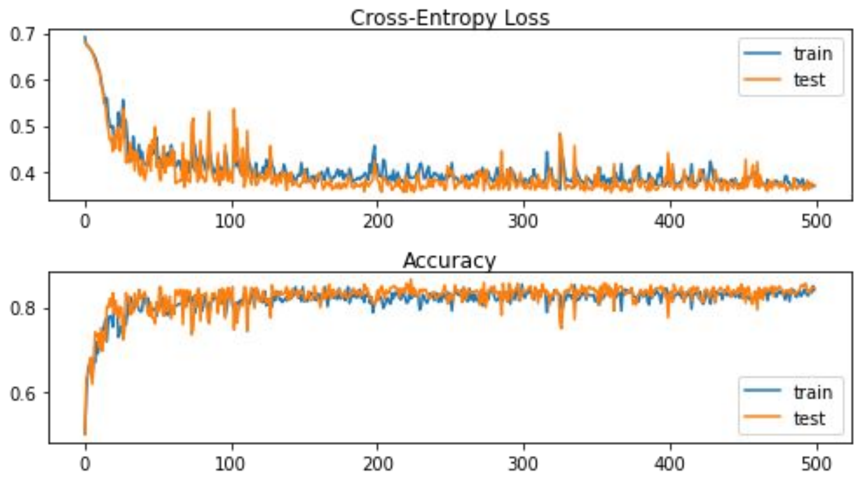


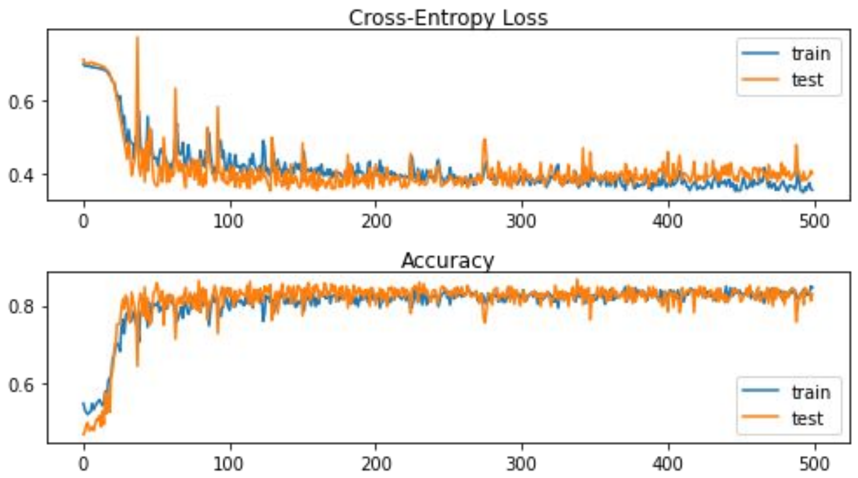
Tanh 5 hidden layer



```
# define model
init = RandomUniform(minval=0, maxval=1)
model = Sequential()
model.add(Dense(5, input_dim=2, activation='relu', kernel_initializer=init))
model.add(Dense(5, activation='relu', kernel_initializer=init))
model.add(Dense(5, activation='relu', kernel_initializer=init))
model.add(Dense(5, activation='relu', kernel_initializer=init))
model.add(Dense(5, activation='relu', kernel_initializer=init))
model.add(Dense(1, activation='sigmoid', kernel_initializer=init))
```

Fix Vanishing Gradients with Relu Case Study





```
# define model
model = Sequential()
model.add(Dense(5, input_dim=2, activation='relu',
kernel_initializer='he_uniform'))
model.add(Dense(5, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(5, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(5, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(5, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(1, activation='sigmoid', kernel_initializer='he_uniform'))
```

```
# define model
model = Sequential()
model.add(Dense(5, input_dim=2, activation='tanh',
kernel_initializer='he_uniform'))
model.add(Dense(5, activation='tanh', kernel_initializer='he_uniform'))
model.add(Dense(5, activation='tanh', kernel_initializer='he_uniform'))
model.add(Dense(5, activation='tanh', kernel_initializer='he_uniform'))
model.add(Dense(5, activation='tanh', kernel_initializer='he_uniform'))
model.add(Dense(1, activation='sigmoid', kernel_initializer='he_uniform'))
```

Fix Vanishing Gradients with Relu Case Study [Extensions]

- **Weight Initialization**: update the deep MLP with tanh activation to use Xavier uniform weight initialization and report the results.
- **Learning Algorithm**: update the deep MLP with tanh activation to use an adaptive learning algorithm such as Adam and report the results.
- **Gradient visualization**: update the tanh and relu examples to record and visualize gradients using TensorBoard for each epoch as a proxy for how much each layer is changed during training and compare results.
- **Study Model Depth**: create an experiment using the MLP with tanh activation and
- report the performance of models as the number of hidden layers is increased from 1 to 10.
- **Increase Breadth**: increase the number of nodes in the hidden layers of the MLP with tanh activation from 5 to 25 and report performance as the number of layers are increased from 1 to 10.

Fix Exploding Gradients with Grad. Clipping

The **underflow** or **overflow** of weights is generally referred to as an instability of the network training process and is known by the name **exploding gradients** as the unstable training process causes the network to fail to train in such a way that the model is essentially useless.

It can happen due to a poor configuration choice. Some examples include:

- **Poor choice of learning rate** that results in large weight updates.
- **Poor choice of data preparation**, allowing large differences in the target variable.
- **Poor choice of the loss function**, allowing the calculation of large error values.

Fix Exploding Gradients with Grad. Clipping

A **common solution to exploding gradients** is to change the error derivative before propagating it backward through the network and using it to update the weights. By **rescaling the error derivative**, the updates to the weights will also be rescaled, dramatically decreasing the likelihood of an overflow or underflow. There are two main methods for updating the error derivative.

Gradient Scaling vs Gradient Clipping

Fix Exploding Gradients with Grad. Clipping

```
# configure sgd with gradient norm clipping
opt = SGD(lr=0.01, momentum=0.9, clipnorm=1.0)
```

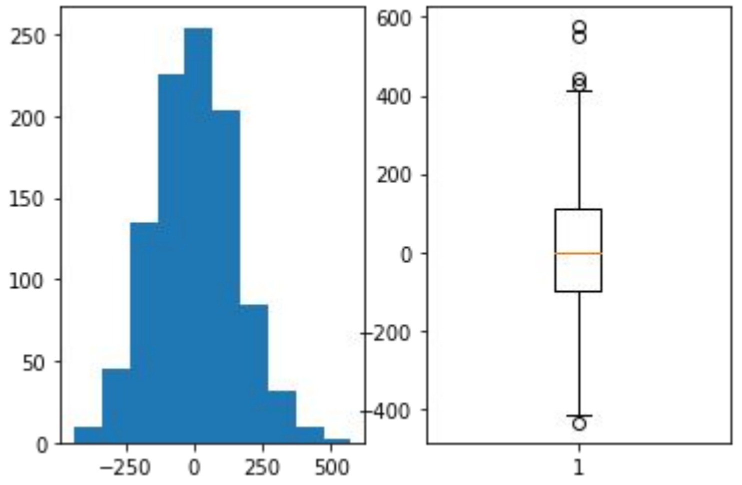
Gradient norm scaling involves changing the loss function derivatives to have a given vector norm when the L2 vector norm (sum of the squared values) of the gradient vector exceeds a threshold value.

```
# configure sgd with gradient value clipping
opt = SGD(lr=0.01, momentum=0.9, clipvalue=0.5)
```

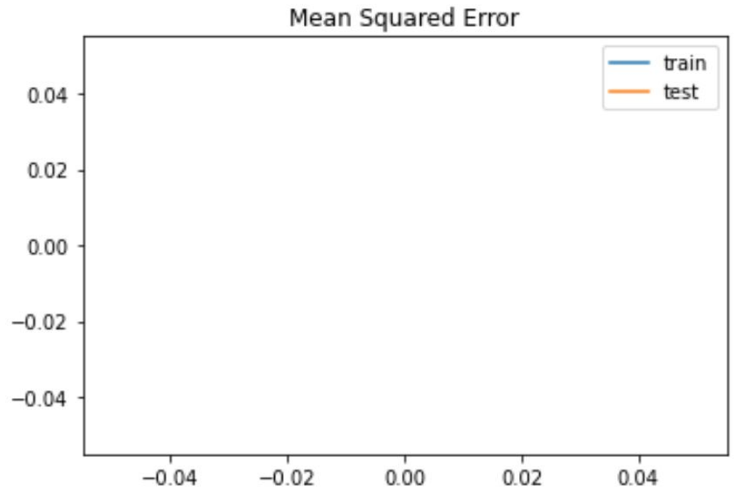
Gradient value clipping involves clipping the loss function's derivatives to have a given value if a gradient value is less than a negative threshold or more than the positive threshold.

Fix Exploding Gradients with Grad. Clipping

MLP with unscaled data

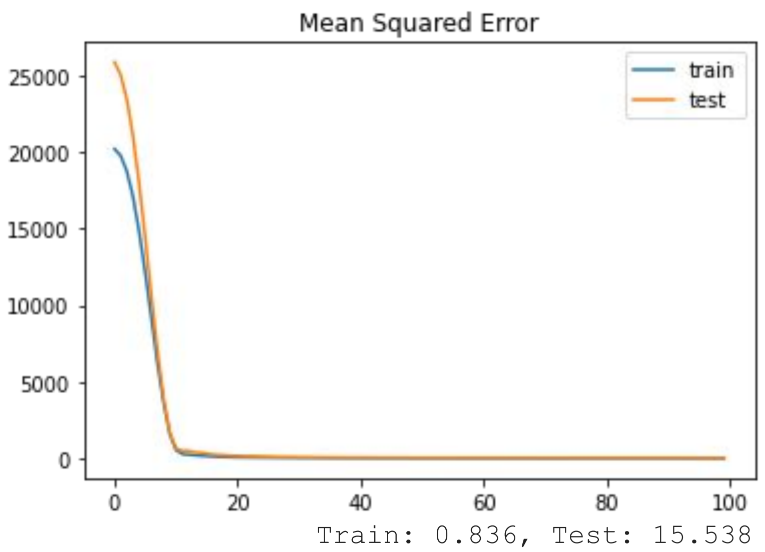


```
Train: nan, Test: nan
```

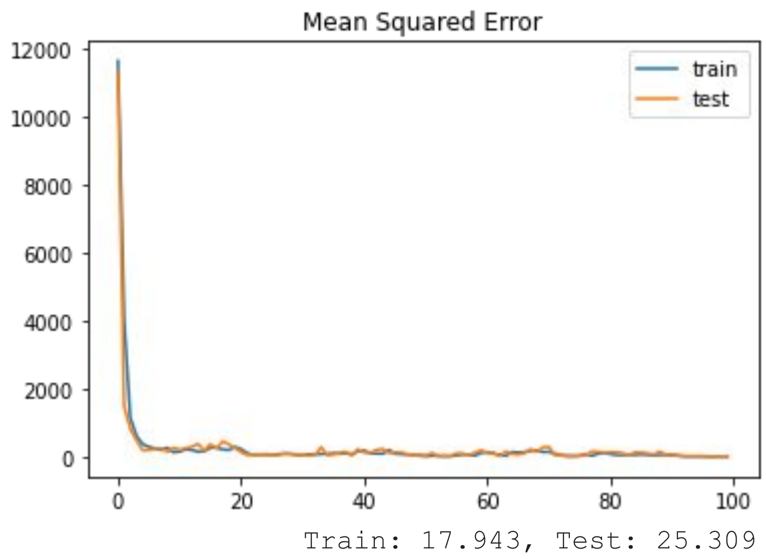


```python
# define model
model = Sequential()
model.add(Dense(25, input_dim=20,
               activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(1, activation='linear'))
```

Fix Exploding Gradients with Grad. Clipping



Train: 0.836, Test: 15.538



Train: 17.943, Test: 25.309

```
# configure sgd with gradient norm clipping
opt = SGD(lr=0.01, momentum=0.9, clipnorm=1.0)
```

```
# configure sgd with gradient value clipping
opt = SGD(lr=0.01, momentum=0.9, clipvalue=0.5)
```

Fix Exploding Gradients with Grad. Clipping

[Extensions]

- **Vector Norm Values**: update the example to evaluate different gradient vector norm values and compare performance.
- **Vector Clip Values**: update the example to evaluate different gradient value ranges and compare performance.
- **Vector Norm and Clip**: update the example to use a combination of vector norm scaling and vector value clipping on the same training run and compare performance.

Next ....

Better Deep Learning

Better Generalization vs Better Learning