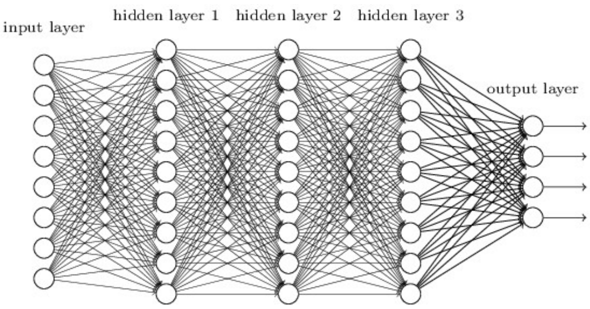
ECE 046211 - Technion - Deep Learning

Tal Daniel

Tutorial 05 - Multilayer Neural Networks



Agenda

Additional Packeges for Google Colab

If you are using Google Colab, you have to install additional packages. To do this, simply run the following cell.

In [ ]:
```
# to work locally (win/linux/mac), first install 'graphviz': https://graphviz.org/download/ and restart your mach
!pip install torchviz
```

In [1]:
```
# imports for the tutorial
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
from torch.utils.data import TensorDataset, DataLoader
import torchviz
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
# %matplotlib notebook
%matplotlib inline
```

Multi-Layer Perceptron (MLP)

- An MLP is composed of one input layer, one or more hidden layers and a final output layer.
- Every layer, except the output layer, optionally includes a **bias neuron** which is fully connected to the next layer.
- When the number of hidden layers is larger than 2, the network is usually called a deep neural network (DNN).

- MLPs are trained with the *backpropgation* algorithm, which is composed of two main parts: **forward pass and backward pass**.
    - This **autodiff reverse mode**.
- In the *forward pass*, for each training instance, the algorithm feeds it to the network and computes the output of every neuron in each consecutive layer (using the network for prediction is just doing a forward pass).
- Then, the output error (the difference between the desired output and the actual output from the network), which is task dependent, is computed.

- After the output error calculation, the network calculates how much each neuron in the last hidden layer contributed to the output error (using the **chain rule**).
- It then proceeds to measure how much of these error contributions came from each neuron in the previous layers until reaching the input layer.
- This is the *backward pass*: measuring the error gradient across all the connection weights in the network by propagating the error gradient backward in the network (this is the backpropagation process).

In short: for each training instance (=batch) the **backpropagation algorithm** first makes a prediction (forward pass), measures the error, then goes in reverse to measure the error contribution from each connection (backward pass) and finally, using Gradient Descent, updates the weights in the direction that reduces the error.

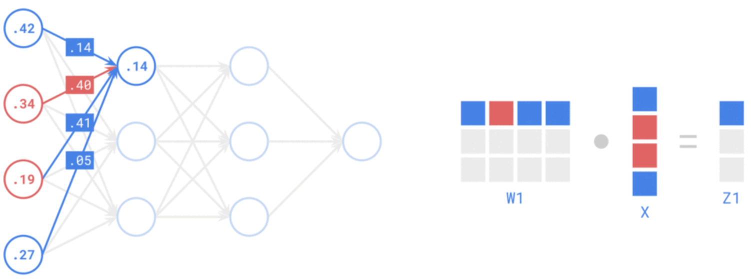
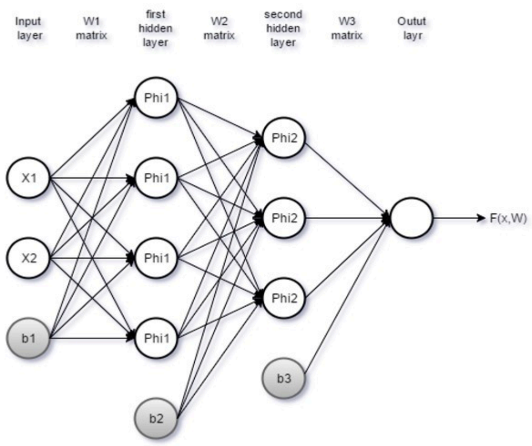


Image Source



For example, if:

$$X \in \mathbb{R}^2$$

$$W_1 \in \mathbb{R}^{2\times 4}$$

$$W_2 \in \mathbb{R}^{4\times 3}$$

$$W_3 \in \mathbb{R}^{3\times 1}$$

$$b_1 \in \mathbb{R}^4$$

$$b_2 \in \mathbb{R}^3$$

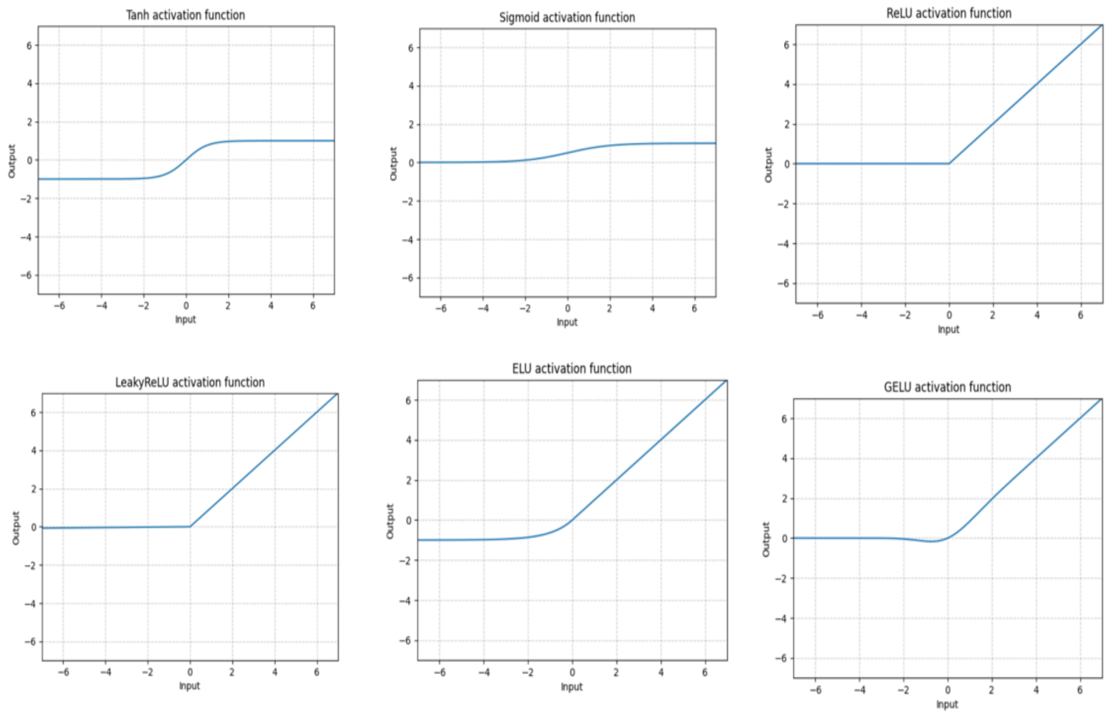
$$b_3 \in \mathbb{R}$$

Then:

$$F(X,W) = W_3^T \phi_2(W_2^T \phi_1(W_1^T X + b_1) + b_2) + b_3$$

The key change made to the Perceptron that brought upon the era of deep learning is the addition of **activation function** to the output of each neuron. These allow the learning of non-linear functions. Some popular activation functions:

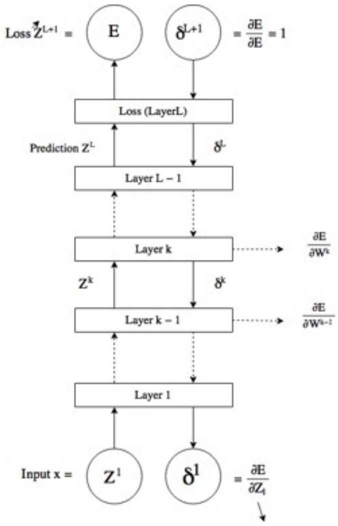
1. **Logistic function (sigmoid)**: $\sigma(z) = \frac{1}{1+e^{-z}}$. The output is in $[0, 1]$ which can be used for binary classification or as a probability. In PyTorch: `nn.Sigmoid()` or `torch.sigmoid()` .
2. **Hyperbolic tangent function**: $tanh(z) = 2\sigma(2z) - 1$. The output is in $[-1, 1]$ which tends to make each layer's output more or less normalized at the beginning of the training (which may speed up convergence). In PyTorch: `nn.Tanh()` or `torch.tanh()` .
3. **ReLU (Rectified Linear Unit) function**: $ReLU(z) = \max(0, z)$. Continuous but not differentiable at $z = 0$. However, it is the most common activation function as it is fast to compute and does not bound the output (which helps with some issues during Gradient Descent). In PyTorch: `nn.ReLU()` or `torch.relu()` .
4. **LeakyReLU function**: $LeakyReLU(z) = \max(0, x) + \text{negative-slope} * \min(0, x)$. An activation function based on a ReLU, but it has a small slope for negative values instead of a flat slope. The slope coefficient is a hyper-parameter determined before training. This type of activation function is popular in tasks where we we may suffer from sparse gradients. In PyTorch: `nn.LeakyReLU(negative_slope=0.01)` .
   - Also see **Exponential Linear Unit (ELU)**: In PyTorch: `nn.ELU(alpha=1.0)` .
5. **Gaussian Error Linear Units (GELU) function**: $GELU(x) = x * \Phi(x)$, where $\Phi(x)$ is the Cumulative Distribution Function (CDF) for the standard Gaussian Distribution. In PyTorch: `nn.GELU()` . Read More.
6. **Sigmoid Linear Unit (SiLU) function**: $silu(x) = x * \sigma(x)$. In PyTorch: `nn.SiLU()` .



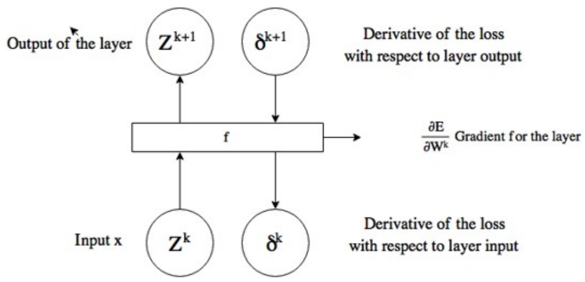
Modular Approach - Autodiff Reverse Mode

- We code **layers**, not networks.
- Layer Specification - each layer needs to provide 3 functions:
  1. The layer output given its input (forward pass) $Z^{(k+1)} = f(Z^{(k)})$.
  2. Derivative with respect to the input $\frac{\partial Z^{(k+1)}}{\partial Z^{(k)}}$.
  3. Derivative with respect to parameters $\frac{\partial Z^{(k+1)}}{\partial W^{(k)}}$.

Illustration:



Zoom-in:



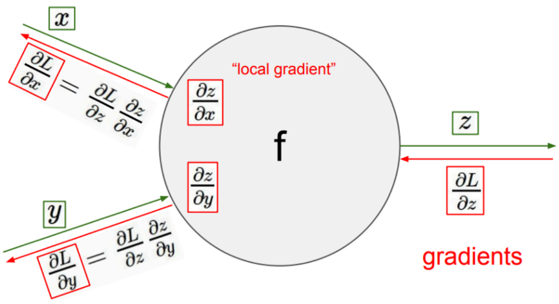


Image source: CS231n Lecture 4

Backpropagation

We now establish a common language when it comes to neural networks architecture (assume single dimension):

- **Forward Pass**: $Z^{(k+1)} = f(Z^{(k)})$
- **Backward Pass**: $\delta^{(k+1)} = \frac{\partial E}{\partial Z^{(k+1)}}$
- Applying the **chain rule** for a single layer:

$$\frac{\partial E}{\partial Z^{(k)}} = \frac{\partial E}{\partial Z^{(k+1)}} \frac{\partial Z^{(k+1)}}{\partial Z^{(k)}} = \delta^{(k+1)} \frac{\partial Z^{(k+1)}}{\partial Z^{(k)}} = \delta^{(k+1)} \frac{\partial f(Z^{(k)})}{\partial Z^{(k)}}$$

- The **gradient with respect to layer parameters** (if it has any):

$$\frac{\partial E}{\partial W^{(k)}} = \frac{\partial E}{\partial Z^{(k+1)}} \frac{\partial Z^{(k+1)}}{\partial W^{(k)}} = \delta^{(k+1)} \frac{\partial Z^{(k+1)}}{\partial W^{(k)}}$$

- **Important Note**: in the above, for multi-dimensional tensors, there is an abuse of dimensions above: $\frac{\partial Z^{(k+1)}}{\partial W^{(k)}}$ is not the proper way of getting the derivative of "a vector w.r.t. a matrix". See explanation under the **The Linear Layer** section below.

Extension to Multi-Dimensions

- $f: \mathbb{R}^n \to \mathbb{R}^m$ is a vector function of a vector variable:

$$f(x) = \begin{bmatrix} f_1(x) \\ \vdots \\ f_m(x) \end{bmatrix}, x \in \mathbb{R}^n, f(x) \in \mathbb{R}^m$$

- The **gradient** is given by:

$$\frac{\partial f_i}{\partial x} = [\frac{\partial f_i(x)}{\partial x_1}, \dots, \frac{\partial f_i(x)}{\partial x_n}]$$

- The **Jacobian**, $J_f(x) \in \mathbb{R}^{m \times n}$, is given by:

$$J_f(x) = \begin{bmatrix} \frac{\partial f_1(x)}{\partial x} \\ \vdots \\ \frac{\partial f_m(x)}{\partial x} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \dots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

- **The Chain Rule**:
  - Given:

$$F: \mathbb{R}^n \to \mathbb{R}^m$$
$$\phi: \mathbb{R}^m \to \mathbb{R}^k$$
$$\psi(x) = \phi(F(x))$$

  - The Jacobian is given by:

$$J_\psi = J_\phi J_F$$
$$J_\phi \in \mathbb{R}^{k \times m}, J_F \in \mathbb{R}^{m \times n} \to J_\psi \in \mathbb{R}^{k \times n}$$

  - Recall that `autograd` implements reverse mode Autodiff as a vector-Jacobian multiplication engine.

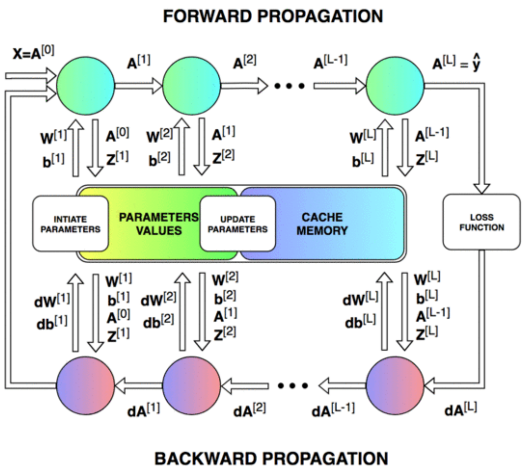


Image Source

Commonly Used Layers (as Modular Blocks)

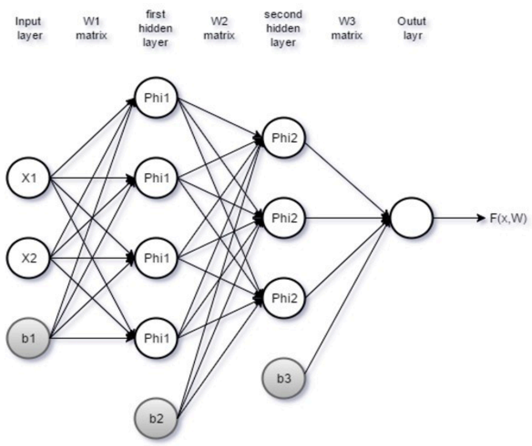
- Linear Layer (linear combination of the inputs).
- Activation Layer (usually together with the linear layer, apply a function on the linear combination of weighted inputs): ReLU, Binary Step, Sigmoid, TanH and etc...
- Softmax Layer (Sigmoid for more than 2 classes, outputs the probability of each class) for classification tasks.

- Loss Function Layer (e.g., MSE and Cross Entropy).

Example - Neural Networks for Regression - Housing Prices

- The Housing Prices Dataset:
  - Two input features: *Size* and *Floor*
  - One output: *House Price*
  - **Loss function**: MSE
- Suggested **network architecture**: 2 hidden layers
  - Two inputs, one for each feature
  - Four neurons in the *first hidden layer*
  - Three neurons in the *second hidden layer*
  - One output

Layout:



$$F(X,W) = W_3^T\phi_2(W_2^T\phi_1(W_1^TX + b_1) + b_2) + b_3$$

Where:

$$X \in \mathbb{R}^2$$

$$W_1 \in \mathbb{R}^{2\times4}$$

$$W_2 \in \mathbb{R}^{4\times3}$$

$$W_3 \in \mathbb{R}^{3\times1}$$

$$b_1 \in \mathbb{R}^4$$

$$b_2 \in \mathbb{R}^3$$

$$b_3 \in \mathbb{R}$$

Step-by-Step Solution

- The MSE loss function over all the training examples x_i and the corresponding training targets:

$$Error = \frac{1}{N}\sum_{i=1}^{N}(F(x_i,W) - y_i)^2 = \frac{1}{N}||F(X,W) - Y||_2^2$$

- Linear Layer:

$$u_{out} = W^T u_{in} + b$$

- Activation Layer:

- ϕ_1 and ϕ_2 are multivariate vector *nonlinear* functions, such that:

$$\phi(U) = \phi\left(\begin{bmatrix} u_1 \\ \vdots \\ u_n \end{bmatrix}\right) = \begin{bmatrix} \phi(u_1) \\ \vdots \\ \phi(u_n) \end{bmatrix}$$

- For **ReLU**:

$$\begin{bmatrix} \phi(u_1) \\ \vdots \\ \end{bmatrix} = \begin{bmatrix} \max(0, u_1) \\ \vdots \\ \end{bmatrix}$$

The Linear Layer

- **Forward Pass**:

$$Z^{(k+1)} = f(Z^{(k)}) = (W^{(k)})^T Z^{(k)} + b^{(k)}.$$

  - k denotes the k^{th} layer with the corresponding weights and bias $W^{(k)}, b^{(k)}$.

- **Derivative** with respect to *input* $Z^{(k)}$:

$$\frac{\partial Z^{(k+1)}}{\partial Z^{(k)}} = \frac{\partial((W^{(k)})^T Z^{(k)} + b^{(k)})}{\partial Z^{(k)}} = (W^{(k)})^T,$$

$$\delta^{(k)} = \delta^{(k+1)}(W^{(k)})^T.$$

- **Derivative** with respect to the *parameters* $W^{(k)}, b^{(k)}$:

$$\frac{\partial Z^{(k+1)}}{\partial W^{(k)}} = Z^{(k)}, \frac{\partial E}{\partial W^{(k)}} = Z^{(k)}\delta^{(k+1)^T},$$

$$\frac{\partial Z^{(k+1)}}{\partial b^{(k)}} = I, \frac{\partial E}{\partial b^{(k)}} = \delta^{(k+1)}.$$

- **Important Note**: in the above, we *abused* the dimensions, e.g., $\frac{\partial Z^{(k+1)}}{\partial W^{(k)}} = Z^{(k)}$ is not the proper way of getting the derivative of "a vector w.r.t. a matrix". The proper way of doing this calculation uses Kronecker product: $\frac{\partial (Ax)}{\partial A} = x \otimes I_m$, where I_m is the $m \times m$ identity matrix. However, since we are starting our calculation from a scalar (our loss), the computation is simplified to a simple *outer product*: $\frac{\partial E}{\partial W^{(k)}} = Z^{(k)}\delta^{(k+1)^T}$.
  - "Derivative of a vector with respect to a matrix" @ math.stackexchange.com.
  - Vector, Matrix, and Tensor Derivatives - Erik Learned-Miller - page 4.
- **Tip**: when calculating backpropagation by-hand, it is always good to perform calculations parameter-wise (i.e., calculate the gradient per parameter and not per layer) to avoid dimensionality issues.

The ReLU Layer

- **Forward Pass**:

$$Z^{(k+1)} = \begin{bmatrix} max(0, Z_1^{(k)}) \\ \vdots \\ max(0, Z_n^{(k)}) \end{bmatrix}, ReLU(Z): \mathbb{R}^n \to \mathbb{R}^n$$

- **Derivative** with respect to *input* $Z^{(k)}$:

$$\phi = max(0, Z^{(k)}), \phi' = heaviside(Z^{(k)})$$

$$\frac{\partial Z^{(k+1)}}{\partial Z^{(k)}} = diag(\phi')$$

$$\delta^{(k)} = \delta^{(k+1)} diag(\phi')$$

- **Derivative** with respect to the *parameters*: **NO PARAMETERS!**

The MSE Layer

- **Forward Pass**:

$$E = Z^{(k+1)} = ||Z^{(k)} - y||^2$$

  - For simplicity, we omit the $\frac{1}{N}$ factor before the MSE term.
- **Derivative** with respect to *input* $Z^{(k)}$:

$$\delta^{(k+1)} = \frac{\partial E}{\partial Z^{(k+1)}} = \frac{\partial E}{\partial E} = 1$$

$$\frac{\partial Z^{(k+1)}}{\partial Z^{(k)}} = 2(Z^{(k)} - y)$$

$$\delta^{(k)} = \delta^{(k+1)} 2(Z^{(k)} - y) = 2(Z^{(k)} - y)$$

Forward Pass

$$F(X, W) = W_3^T \phi_2(W_2^T \phi_1(W_1^T X + b_1) + b_2) + b_3$$

- Input
- $Z^{(0)} = x$
- Linear layer
- $Z^{(1)} = W_1^T Z^{(0)} + b_1$
- Activation layer
- $Z^{(2)} = \varphi_1(Z^{(1)})$
- Linear layer
- $Z^{(3)} = W_2^T Z^{(2)} + b_2$
- Activation layer
- $Z^{(4)} = \varphi_2(Z^{(3)})$
- Linear layer
- $Z^{(5)} = W_3^T Z^{(4)} + b_3$
- Loss layer
- $Z^{(6)} = \left(Z^{(5)} - y\right)^2$

Backward Pass

The following illustration depicts the backpropagation process:

- Input
- $Z^{(0)} = x$
- Linear layer
- $Z^{(1)} = W_1^T Z^{(0)} + b_1$
- Activation layer
- $Z^{(2)} = \varphi_1(Z^{(1)})$
- Linear layer
- $Z^{(3)} = W_2^T Z^{(2)} + b_2$
- Activation layer
- $Z^{(4)} = \varphi_2(Z^{(3)})$
- Linear layer
- $Z^{(5)} = W_3^T Z^{(4)} + b_3$
- Loss layer
- $Z^{(6)} = \left(Z^{(5)} - y\right)^2$

$$\delta_0 = \delta_1 \cdot W_1^T \qquad \frac{\partial E}{\partial W_1} = Z^{(0)} \delta_1^T \qquad \frac{\partial E}{\partial b_1} = \delta_1$$

$$\delta_1 = \delta_2 \cdot diag\left(heaviside\left(Z^{(1)}\right)\right)$$

$$\delta_2 = \delta_3 \cdot W_2^T \qquad \frac{\partial E}{\partial W_2} = Z^{(2)} \delta_3^T \qquad \frac{\partial E}{\partial b_2} = \delta_3$$

$$\delta_3 = \delta_4 \cdot diag\left(heaviside\left(Z^{(3)}\right)\right)$$

$$\delta_4 = \delta_5 \cdot W_3^T \qquad \frac{\partial E}{\partial W_3} = Z^{(4)} \delta_5^T \qquad \frac{\partial E}{\partial b_3} = \delta_5$$

$$\delta_5 = \delta_6 \cdot 2\left(Z^{(5)} - y\right)$$

$$\delta_6 = 1$$

- Remember that `autograd` is a vector-Jacobian multiplication engine.

Building a Neural Network with PyTorch

We will now implement a neural network for regression with PyTorch. We will use the "Boston House Prices" dataset and use the architecture described above.

In [2]:
```python
# define our neural network model
# this approach provides easier access to weights (e.g., 'model.fc1' will return the parameters of the first laye
class HousePricesMLP(nn.Module):
    # notice that we inherit from nn.Module
    def __init__(self, input_dim, output_dim):
        super(HousePricesMLP, self).__init__()
        # here we initialize the building blocks of our network
        # single neuron is just one linear (fully-connected) layer
        self.fc_1 = nn.Linear(input_dim, 4)
        self.fc_2 = nn.Linear(4, 3)
        self.output_layer = nn.Linear(3, output_dim)

    def forward(self, x):
        # here we define what happens to the input x in the forward pass
        # that is, the order in which x goes through the building blocks
        x = torch.relu(self.fc_1(x))
        x = torch.relu(self.fc_2(x))
        return self.output_layer(x)
```

In [3]:
```python
# alternative method - more readdable, easier to code, less convenient access to weights
# e.g., to access the first layer weights -- `model.hidden[0]`
class HousePricesMLP(nn.Module):
    # notice that we inherit from nn.Module
    def __init__(self, input_dim, output_dim):
        super(HousePricesMLP, self).__init__()
        # here we initialize the building blocks of our network
        # single neuron is just one linear (fully-connected) layer
        self.hidden = nn.Sequential(nn.Linear(input_dim, 4),
                                    nn.ReLU(),
                                    nn.Linear(4, 3),
                                    nn.ReLU())
        self.output_layer = nn.Linear(3, output_dim)

    def forward(self, x):
        # here we define what happens to the input x in the forward pass
        # that is, the order in which x goes through the building blocks
        return self.output_layer(self.hidden(x))
```

In [3]:
```python
# NOTE: in this example we are using a very simple NN model
# We usually wider and deeper networks such as this one:
class HousePricesMLP(nn.Module):
    # notice that we inherit from nn.Module
    def __init__(self, input_dim, output_dim, hidden_dim=256):
```

```
        super(HousePricesMLP, self).__init__()
        # here we initialize the building blocks of our network
        # single neuron is just one linear (fully-connected) layer
        self.hidden = nn.Sequential(nn.Linear(input_dim, hidden_dim),
                                    nn.ReLU(),
                                    nn.Linear(hidden_dim, hidden_dim),
                                    nn.ReLU(),
                                    nn.Linear(hidden_dim, hidden_dim),
                                    nn.ReLU(),)
        self.output_layer = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        # here we define what happens to the input x in the forward pass
        # that is, the order in which x goes through the building blocks
        return self.output_layer(self.hidden(x))
```

In [4]:

```
# load data and preprocess
boston_dataset = load_boston()
# print description of the features
print(boston_dataset.DESCR)
```

```
.. _boston_dataset:

Boston house prices dataset
---------------------------

**Data Set Characteristics:**

    :Number of Instances: 506

    :Number of Attributes: 13 numeric/categorical predictive. Median Value (attribute 14) is usually the target.

    :Attribute Information (in order):
        - CRIM     per capita crime rate by town
        - ZN       proportion of residential land zoned for lots over 25,000 sq.ft.
        - INDUS    proportion of non-retail business acres per town
        - CHAS     Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
        - NOX      nitric oxides concentration (parts per 10 million)
        - RM       average number of rooms per dwelling
        - AGE      proportion of owner-occupied units built prior to 1940
        - DIS      weighted distances to five Boston employment centres
        - RAD      index of accessibility to radial highways
        - TAX      full-value property-tax rate per $10,000
        - PTRATIO  pupil-teacher ratio by town
        - B        1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town
        - LSTAT    % lower status of the population
        - MEDV     Median value of owner-occupied homes in $1000's

    :Missing Attribute Values: None

    :Creator: Harrison, D. and Rubinfeld, D.L.

This is a copy of UCI ML housing dataset.
https://archive.ics.uci.edu/ml/machine-learning-databases/housing/


This dataset was taken from the StatLib library which is maintained at Carnegie Mellon University.

The Boston house-price data of Harrison, D. and Rubinfeld, D.L. 'Hedonic
prices and the demand for clean air', J. Environ. Economics & Management,
vol.5, 81-102, 1978.   Used in Belsley, Kuh & Welsch, 'Regression diagnostics
...', Wiley, 1980.   N.B. Various transformations are used in the table on
pages 244-261 of the latter.

The Boston house-price data has been used in many machine learning papers that address regression
problems.

.. topic:: References

   - Belsley, Kuh & Welsch, 'Regression diagnostics: Identifying Influential Data and Sources of Collinearity', W
iley, 1980. 244-261.
   - Quinlan,R. (1993). Combining Instance-Based and Model-Based Learning. In Proceedings on the Tenth Internatio
nal Conference of Machine Learning, 236-243, University of Massachusetts, Amherst. Morgan Kaufmann.
```

In [5]:

```
# the target is the MEDV field - median value of owner-occupied homes in 1000$
boston = pd.DataFrame(boston_dataset.data, columns=boston_dataset.feature_names)
boston['MEDV'] = boston_dataset.target
boston.sample(10)
```

Out[5]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
10	0.22489	12.5	7.87	0.0	0.524	6.377	94.3	6.3467	5.0	311.0	15.2	392.52	20.45	15.0
220	0.35809	0.0	6.20	1.0	0.507	6.951	88.5	2.8617	8.0	307.0	17.4	391.70	9.71	26.7
422	12.04820	0.0	18.10	0.0	0.614	5.648	87.6	1.9512	24.0	666.0	20.2	291.55	14.10	20.8
296	0.05372	0.0	13.92	0.0	0.437	6.549	51.0	5.9604	4.0	289.0	16.0	392.85	7.39	27.1
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14	21.6
351	0.07950	60.0	1.69	0.0	0.411	6.579	35.9	10.7103	4.0	411.0	18.3	370.78	5.49	24.1
261	0.53412	20.0	3.97	0.0	0.647	7.520	89.4	2.1398	5.0	264.0	13.0	388.37	7.26	43.1
347	0.01870	85.0	4.15	0.0	0.429	6.516	27.7	8.5353	4.0	351.0	17.9	392.43	6.36	23.1
175	0.06664	0.0	4.05	0.0	0.510	6.546	33.1	3.1323	5.0	296.0	16.6	390.96	5.33	29.4
416	10.83420	0.0	18.10	0.0	0.679	6.782	90.8	1.8195	24.0	666.0	20.2	21.57	25.79	7.5

In [6]:

```
# we will use 2 features
x = boston[['RM', 'LSTAT']].values  # RM-num rooms, LSTAT-% lower status of the population
y = boston['MEDV'].values
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.2, random_state=5)
# scaling
x_scaler = StandardScaler()
x_scaler.fit(x_train)
x_train = x_scaler.transform(x_train)
x_test = x_scaler.transform(x_test)
print("total training samples: {}, total test samples: {}".format(len(x_train),len(x_test)))
```

```
total training samples: 404, total test samples: 102
```

In [7]:

```
# convert to tensor dataset for PyTorch
# boston_tensor_train_ds = TensorDataset(torch.from_numpy(x_train).float(), torch.from_numpy(y_train).float()) #
boston_tensor_train_ds = TensorDataset(torch.tensor(x_train, dtype=torch.float), torch.tensor(y_train, dtype=torc
# check
print(f'sample 0: features: {boston_tensor_train_ds[0][0]}, target: {boston_tensor_train_ds[0][1]}')
```

```
sample 0: features: tensor([-0.8488,  0.8353]), target: 13.100000381469727
```

In [8]:

```
# define hyper-parmeters and create our model
num_features = 2
output_dim = 1
batch_size = 128
learning_rate = 0.01
num_epochs = 500
# device
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
# loss criterion
criterion = nn.MSELoss()
# model
model = HousePricesMLP(num_features, output_dim).to(device)
# optimizer
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

In [27]:

```
boston_tensor_train_dataloader = DataLoader(boston_tensor_train_ds, batch_size=batch_size, shuffle=True)

# training loop for the model
for epoch in range(num_epochs):
    model.train()  # change the mode to training, activating layers like DropOut and BatchNorm, if there are any
    epoch_losses = []
    for features, targets in boston_tensor_train_dataloader:
        # send data to device
        features = features.to(device)
        targets = targets.to(device)
        # forward pass
        output = model(features)  # calls model.forward(features)
        # loss
        loss = criterion(output.view(-1), targets)
        # backward pass
        optimizer.zero_grad()  # clean the gradients from previous iteration, clears the `tensor.grad` field (ter
        loss.backward()  # autograd backward to calculate gradients, assigns the `tensor.grad` field (e.g., tens
        optimizer.step()  # apply update to the weights, applies the gradient update rule of the optimizer (param
        epoch_losses.append(loss.item())
    if epoch % 50 == 0:
        print(f'epoch: {epoch} loss: {np.mean(epoch_losses)}')
```

```
epoch: 0 loss: 611.3372344970703
epoch: 50 loss: 26.573792934417725
epoch: 100 loss: 23.42008924484253
epoch: 150 loss: 22.77083158493042
epoch: 200 loss: 22.498516082763672
epoch: 250 loss: 22.412588596343994
epoch: 300 loss: 22.351622104644775
epoch: 350 loss: 22.29261350631714
epoch: 400 loss: 22.230255603790283
epoch: 450 loss: 22.17220449447632
```

In [28]:

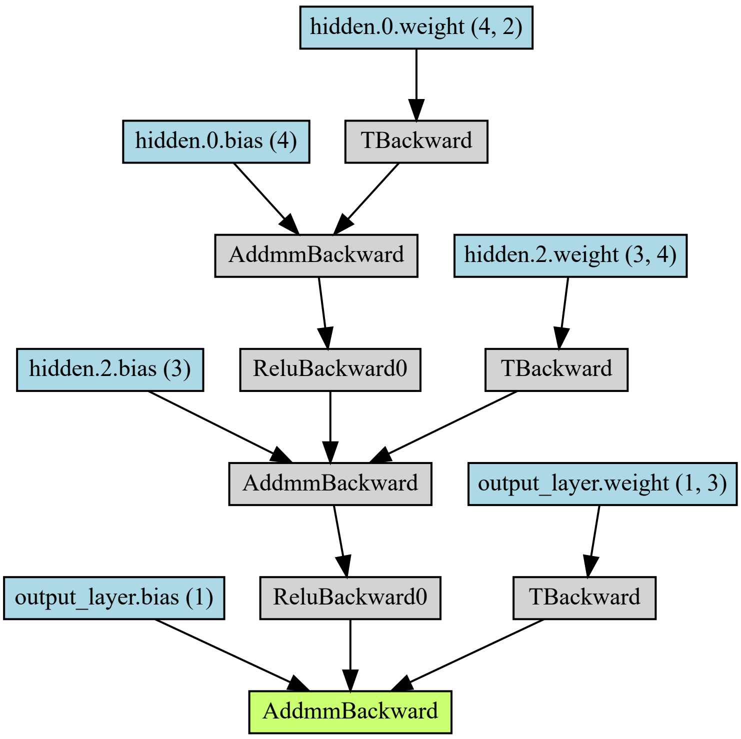
```python
# test error
model.eval()  # put the model in evaluation mode, turns-off drop-out, changes functionality of BatchNorm
# use model.train() to change back to traininig mode
with torch.no_grad():
    # set requires_grad=False for all tensors (weights and biases)
    # test_outputs = model(torch.from_numpy(x_test).float().to(device))  # old method
    test_outputs = model(torch.tensor(x_test, dtype=torch.float, device=device))
    test_error = criterion(test_outputs.view(-1), torch.tensor(y_test, dtype=torch.float, device=device))
print(f'test MSE error: {test_error.item()}')
```

```
test MSE error: 15.394033432006836
```

In [9]:

```python
# visualize computational graph
x = torch.randn(1, num_features, device=device)
torchviz.make_dot(model(x), params=dict(model.named_parameters()))
```

Out[9]:



Weights Initialization

- As we have learned, neural networks are trained using a stochastic optimization algorithm, such as Gradient Descent, RMSprop, Adam and etc...
- Recall that these algorithms require initializing the parameters to some values. That is, they use randomness in order to find a good enough set of weights for the specific mapping function from inputs to outputs in your data that is being learned.
- These algroithms require that the weights of the network are initialized to small random values (random, but close to zero).
    - Randomness is also used during the search process in the **shuffling of the training dataset** prior to each epoch, which in turn results in differences in the gradient estimate for each batch.
- Training deep models is a sufficiently difficult task that most algorithms are strongly affected by the choice of initialization (page 301, Deep Learning, 2016).

Why Not Just Initialize With Zeros?

- We can use the same set of weights each time we train the network. For example, you could use the values of 0.0 for all weights.
- In this case, the equations of the learning algorithm would fail to make any changes to the network weights, and the model will be **stuck**.
    - It is important to note that the bias weight in each neuron is set to zero by default, not a small random value.
- Specifically, neurons that are in the same hidden layer that is connected to the same inputs must have different weights for the learning algorithm to update the weights.
- **Symmetry Breaking**: initial parameters need to "break symmetry" between different units. If two hidden units with the same activation function are connected to the same inputs, then these units must have different initial parameters (page 301, Deep Learning, 2016).
    - Why? If they have the same initial parameters, then a deterministic learning algorithm applied to a deterministic cost and model will constantly update both of these units in the same way.
- Note that when you **constant the seed**, you will initialize with the same weights each time. We do this when we want to get reproducible results (or in production).

Recent Trend: Non-Random Initializations

- Beyond Signal Propagation: Is Feature Diversity Necessary in Deep Neural Network Initialization?, a deep network is constructed with identical features by initializing almost all the weights to 0. The architecture also enables perfect signal propagation and stable gradients, and achieves high accuracy on standard benchmarks, indicating that random, diverse initializations are not always necessary for training neural networks.
- ZerO Initialization: Initializing Neural Networks with only Zeros and Ones, the random weight initialization is replaced with a fully deterministic initialization scheme which initializes the weights of networks with only zeros and ones (up to a normalization factor), based on identity and Hadamard transforms. They show promising results on various benchmarks, paving the way to simpler initializations schemes that work just as well as random initializations.

Types of Weight Initialization

- The initialization of the weights of neural networks is an active field of study as the careful initialization of the network can speed up the learning process.

- There is no single best way to initialize the weights of a neural network.

- We will review some of the popular initalization methods.

- **Unifrom** - initialize with values drawn from the uniform distribution $\mathcal{U}(a, b)$
    - In PyTorch - `torch.nn.init.uniform_(tensor, a=0.0, b=1.0)`
- **Normal** - initialize with values drawn from the normal distribution $\mathcal{N}(\text{mean}, \text{std}^2)$
    - In PyTorch - `torch.nn.init.normal_(tensor, mean=0.0, std=1.0)`
- **Constant** - initialize with the value val.
    - In PyTorch - `torch.nn.init.constant_(tensor, val)`
- **Ones** - Initialize with the scalar value 1.
    - In PyTorch - `torch.nn.init.ones_(tensor)`
- **Zeros** - Initialize with the scalar value 0.
    - In PyTorch - `torch.nn.init.zeros_(tensor)`

- **Xavier (Glorot) Uniform** - Initialize with values according to the method described in *Understanding the difficulty of training deep feedforward neural networks - Glorot, X. & Bengio, Y. (2010)*, using a uniform distribution. The resulting tensor will have values sampled from $\mathcal{U}(-a, a)$ where

$$a = \text{gain} \times \sqrt{\frac{6}{\text{fan}_{in} + \text{fan}_{out}}}$$

    - `fan_in` is the number of input units in the weight tensor and `fan_out` is the number of output units in the weight tensor
    - In PyTorch - `torch.nn.init.xavier_uniform_(tensor, gain=1.0)`
- **Xavier (Glorot) Normal** - Initialize with values according to the method described in *Understanding the difficulty of training deep feedforward neural networks - Glorot, X. & Bengio, Y. (2010)*, using a normal distribution. The resulting tensor

will have values sampled from $\mathcal{N}(0, \text{std}^2)$ where

$$\text{std} = \text{gain} \times \sqrt{\frac{2}{\text{fan}_{in} + \text{fan}_{out}}}$$

  - `fan_in` is the number of input units in the weight tensor and `fan_out` is the number of output units in the weight tensor
  - In PyTorch - `torch.nn.init.xavier_normal_(tensor, gain=1.0)`

- **Kaiming (He) Uniform** - Initialize with values according to the method described in *Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification - He, K. et al. (2015)*, using a uniform distribution. The resulting tensor will have values sampled from $\mathcal{U}(-\text{bound}, \text{bound})$ where

$$\text{bound} = \text{gain} \times \sqrt{\frac{3}{\text{fan-mode}}}$$

  - In PyTorch - `torch.nn.init.kaiming_uniform_(tensor, a=0, mode='fan_in', nonlinearity='leaky_relu')`
  - `a` - the negative slope of the rectifier used after this layer (only used with `leaky_relu`)
  - `fan_mode` – either `fan_in` (default) or `fan_out`. Choosing `fan_in` preserves the magnitude of the variance of the weights in the forward pass. Choosing `fan_out` preserves the magnitudes in the backwards pass.
  - `nonlinearity` – the non-linear function (`nn.functional` name), recommended to use only with `relu` or `leaky_relu` (default).
- **Kaiming (He) Normal** - Initialize with values according to the method described in *Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification - He, K. et al. (2015)*, using a normal distribution. The resulting tensor will have values sampled from $\mathcal{N}(0, \text{std}^2)$ where

$$\text{std} = \frac{\text{gain}}{\sqrt{\text{fan-mode}}}$$

  - In PyTorch - `torch.nn.init.kaiming_normal_(tensor, a=0, mode='fan_in', nonlinearity='leaky_relu')`

PyTorch has default initializations schemes that usually work good. For example, `kaiming_uniform` is the default initialization in PyTorch for `Linear` layers:

In [ ]:
```python
# pytorch default initialization for linear layers
def reset_parameters(module):
    # Setting a=sqrt(5) in kaiming_uniform is the same as initializing with
    # uniform(-1/sqrt(in_features), 1/sqrt(in_features)). For details, see
    # https://github.com/pytorch/pytorch/issues/57109
    torch.nn.init.kaiming_uniform_(module.weight, a=math.sqrt(5))
    if self.bias is not None:
        fan_in, _ = torch.nn.init._calculate_fan_in_and_fan_out(module.weight)
        bound = 1 / math.sqrt(fan_in) if fan_in > 0 else 0
        torch.nn.init.uniform_(module.bias, -bound, bound)
```

Interactive Demo

Different Initializations Demo

Neural Network Weight Initialization with PyTorch

- As from PyTorch 1.0, **most layers are initialized using Kaiming Uniform method by default**.
- Let's see how we change the initialization of a model.
- Official PyTorch initialization documentation.

In [7]:
```python
# define hyper-parmeters and create our model
num_features = 2
output_dim = 1
batch_size = 128
learning_rate = 0.01
num_epochs = 500
# device
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

```python
# loss criterion
criterion = nn.MSELoss()
# model
model = HousePricesMLP(num_features, output_dim).to(device)
# optimizer
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

In [10]:
```python
# use a different initialization for the model
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Linear') != -1:
        torch.nn.init.xavier_normal_(m.weight, gain=1.0)
model.apply(weights_init)
```

Out[10]:
```
HousePricesMLP(
  (hidden): Sequential(
    (0): Linear(in_features=2, out_features=4, bias=True)
    (1): ReLU()
    (2): Linear(in_features=4, out_features=3, bias=True)
    (3): ReLU()
  )
  (output_layer): Linear(in_features=3, out_features=1, bias=True)
)
```

In [ ]:
```python
# another way to do that
class HousePricesMLP(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(HousePricesMLP, self).__init__()
        self.hidden = nn.Sequential(nn.Linear(input_dim, 4),
                                    nn.ReLU(),
                                    nn.Linear(4, 3),
                                    nn.ReLU())
        self.output_layer = nn.Linear(3, output_dim)
        # NEW: init weights here
        self.init_weights()

    def forward(self, x):
        return self.output_layer(self.hidden(x))

    def init_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Linear):
                torch.nn.init.xavier_normal_(m.weight, gain=1.0)
                if m.bias is not None:
                    torch.nn.init.constant_(m.bias, 0)
```

In [11]:
```python
boston_tensor_train_dataloader = DataLoader(boston_tensor_train_ds, batch_size=batch_size, shuffle=True)

# training loop for the model
for epoch in range(num_epochs):
    epoch_losses = []
    for features, targets in boston_tensor_train_dataloader:
        # send data to device
        features = features.to(device)
        targets = targets.to(device)
        # forward pass
        output = model(features)
        # loss
        loss = criterion(output.view(-1), targets)
        # backward pass
        optimizer.zero_grad()  # clean the gradients from previous iteration
        loss.backward()  # autograd backward to calculate gradients
        optimizer.step()  # apply update to the weights
        epoch_losses.append(loss.item())
    if epoch % 50 == 0:
        print(f'epoch: {epoch} loss: {np.mean(epoch_losses)}')

# test error
model.eval()
with torch.no_grad():
    test_outputs = model(torch.tensor(x_test, dtype=torch.float, device=device))
    test_error = criterion(test_outputs.view(-1), torch.tensor(y_test, dtype=torch.float, device=device))
print(f'test MSE error: {test_error.item()}')
```

```
epoch: 0 loss: 624.5885009765625
epoch: 50 loss: 35.445608139038086
epoch: 100 loss: 28.66716480255127
epoch: 150 loss: 22.615984439849854
epoch: 200 loss: 20.051589488983154
epoch: 250 loss: 19.82143211364746
epoch: 300 loss: 19.730319499969482
epoch: 350 loss: 19.615483283996582
epoch: 400 loss: 19.475394248962402
epoch: 450 loss: 19.32789421081543
test MSE error: 15.348824501037598
```

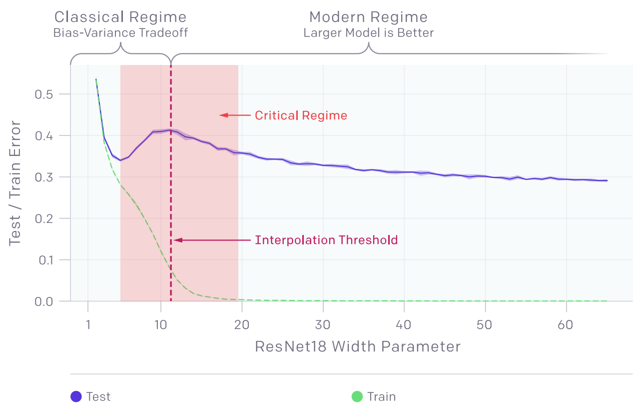
Deep Double Descent

- Double Descent in ML algorithms training: performance first improves, then gets worse, and then improves again with increasing model size, data size, or training time.
- This effect is often avoided through careful **regularization** or **early stopping**.
  - While this behavior appears to be fairly universal, *we don't yet fully understand why it happens*.

- It can be seen that as we increase the number of parameters in a model, the test error initially decreases, increases, and, just as the model is able to fit the train set, undergoes a second descent. This is different than what we saw when we talked about the bias-variance trade-off.
- Double descent also occurs over **train epochs**.
  - Surprisingly, it can lead to a regime where **more data hurts**, and training a deep network on a larger train set actually performs worse.

Model-wise Double Descent

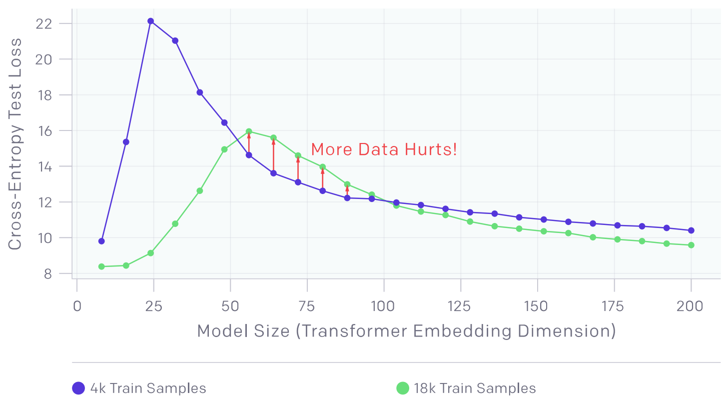
- There is a regime where **bigger models are worse**.
- The model-wise double descent phenomenon can lead to a regime where training on more data hurts.



In the figure, the peak in test error occurs around the interpolation threshold, **when the models are just barely large enough to fit the train set**.

Sample-wise Non-monotonicity

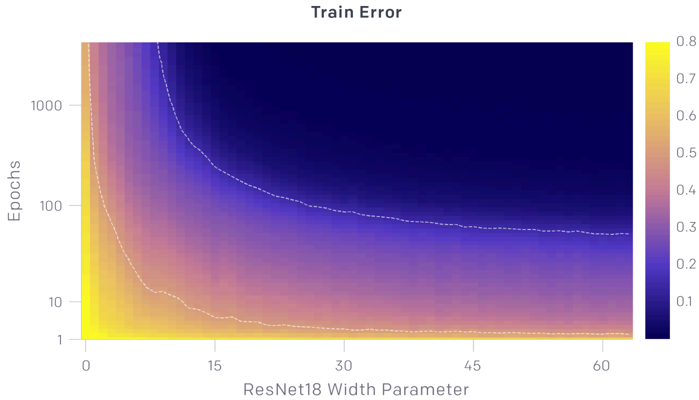
- There is a regime where **more samples hurts**.

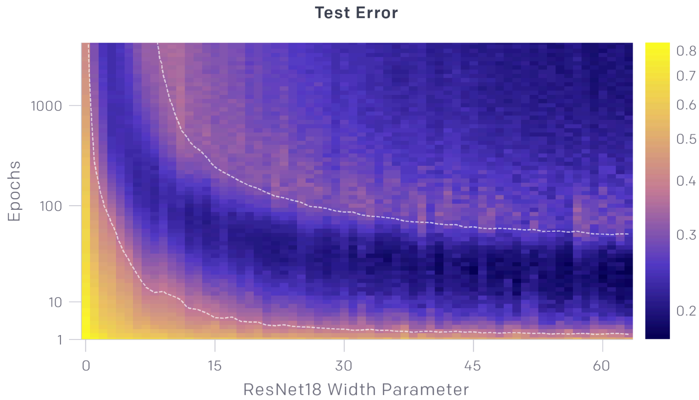


- In the figure, increasing the number of samples shifts the curve downwards towards lower test error.
- However, since more samples require larger models to fit, increasing the number of samples also shifts the interpolation threshold (and peak in test error) to the right.
- For intermediate model sizes (red arrows), these two effects combine, and training on 4.5x more samples actually hurts test performance.

Epoch-wise Double Descent

- There is a regime where **training longer reverses overfitting**.





- The figures above show test and train error as a function of both model size and number of optimization steps.
- For a given number of optimization steps (fixed y-coordinate), test and train error exhibit model-size double descent.
- For a given model size (fixed x-coordinate), as training proceeds, **test and train error decreases, increases, and decreases again!**

In general, the peak of test error appears systematically when models are just barely able to fit the train set.

Recommended Videos

Warning!

- These videos do not replace the lectures and tutorials.
- Please use these to get a better understanding of the material, and not as an alternative to the written material.

Video By Subject

- Deep Learning - Machine Learning Lecture 35 "Neural Networks / Deep Learning" -Cornell CS4780
  - Machine Learning Lecture 36 "Neural Networks / Deep Learning Continued" -Cornell CS4780
- Building a Network with PyTorch - Deep Learning and Neural Networks with Python and Pytorch
- Weight Initialization - UC Berkeley - STAT 157- Stabilize Training - Weight Initialization
  - Krish Naik - Various Weight Initialization Techniques in Neural Network
  - Weight Initialization in a Deep Network (C2W1L11)
- Deep Double Descent - Henry AI Labs - Deep Double Descent

Credits

- Icons made by Becris from www.flaticon.com
- Icons from Icons8.com - https://icons8.com

- Datasets from Kaggle - https://www.kaggle.com/
- Jason Brownlee - Why Initialize a Neural Network with Random Weights?
- OpenAI - Deep Double Descent