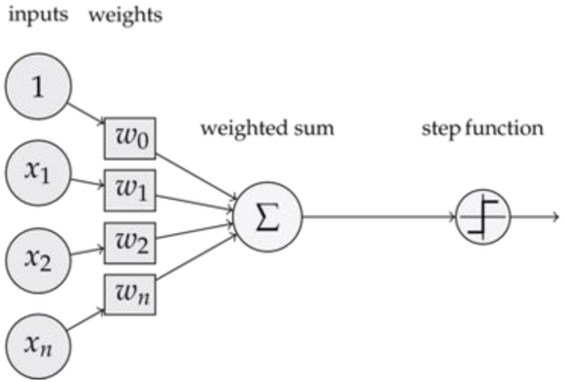
Multilayer perceptrons

Recall from last two lectures

https://blog.knoldus.com/introduction-to-perceptron-neural-network



- Least squares classifier:

  $\hat{\mathbf{y}} = \mathbf{W}\mathbf{x} \qquad L(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{2}\sum_{i=1}^{m}(\mathbf{y}_i - \hat{\mathbf{y}}_i)^2$

- Perceptron:

  $\hat{y} = \mathrm{sign}\left(\mathbf{w}^\top \mathbf{x}\right) \qquad L(y, \hat{y}) = -\sum_{i=1}^{m} \mathbf{w}^\top \mathbf{x}_i y_i$

- Logistic regression:

  $\hat{y} = \sigma(\mathbf{w}^\top \mathbf{x}) \qquad L(y, \hat{y}) = -\sum_{i=1}^{m} y_i \log \sigma(\mathbf{w}^\top \mathbf{x}) + (1 - y_i) \log(1 - \sigma(\mathbf{w}^\top \mathbf{x}))$

Fully connected NNs

- In DL, you will usually see this form:

  $\mathbf{h} = g(\mathbf{W}\mathbf{x} + \mathbf{b})$

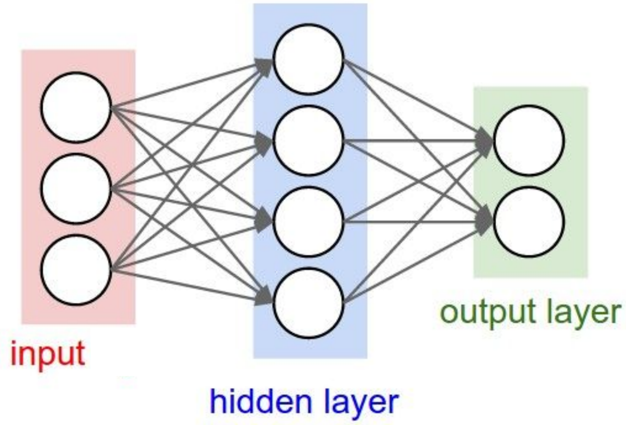
  W: Matrix of weights, one vector per neuron

  x: One input example (vector)

  b: Vector of biases, one scalar per neuron

  h: Hidden layer response

http://cs231n.github.io/neural-networks-1/



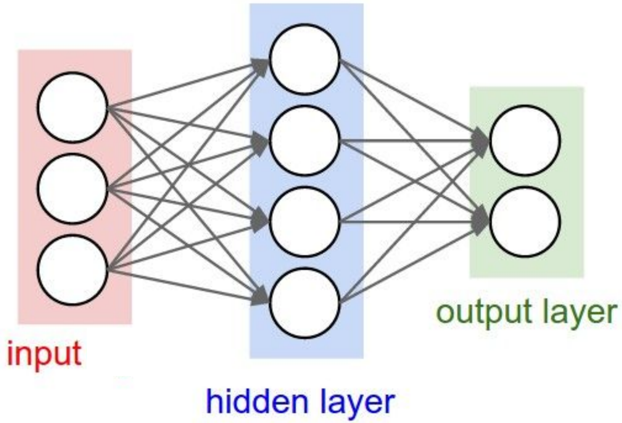
Hyperparameters

- Number of layers
- Propagation types: fully connected, convolutional (later)
- Activation function(s)
- Loss function(s) and parameters (also later)
- Training iterations and batch sizes (some of it today)

Input layer

- A vectorized version of the input data
- Sometimes preprocessed
- Weights connect to the hidden layer

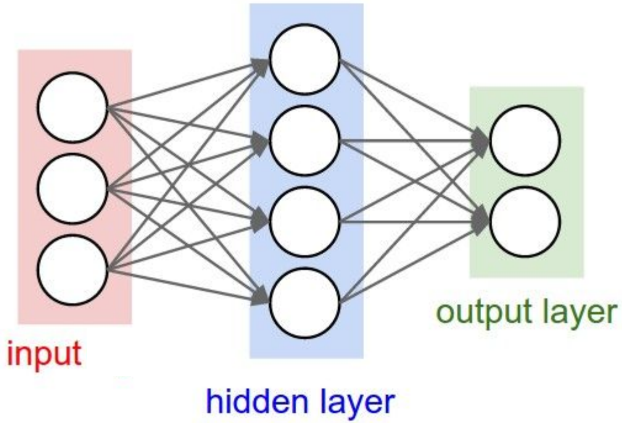
http://cs231n.github.io/neural-networks-1/



Hidden layer(s)

- Number of hidden layers
- Number of neurons
- Topology - expanding or bottleneck
- Strongly application-dependent
- Currently no optimal way to design in advance

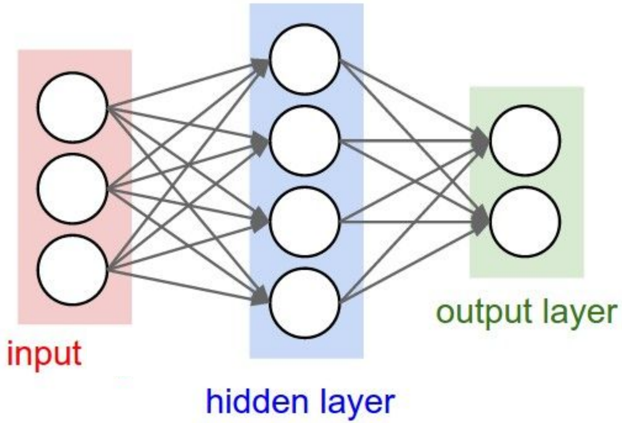
http://cs231n.github.io/neural-networks-1/



Output layer

- For regression:
  - Linear outputs with MSE
- For classification
  - Softmax units (logistic for two classes)
- - and many, many more...

http://cs231n.github.io/neural-networks-1/



Activation functions

$$\boxed{\sigma'(x) = \sigma(x)(1 - \sigma(x))}$$

$$\boxed{\tanh'(x) = 1 - \tanh(x)^2}$$

$$\boxed{\mathrm{relu}'(x) = \mathrm{step}(x)}$$

Sigmoid

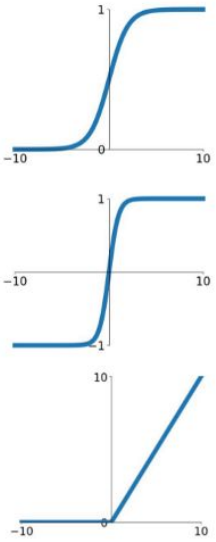
$$\sigma(x) = \frac{1}{1+e^{-x}}$$

tanh

$$\tanh(x)$$

ReLU

$$\max(0, x)$$



https://towardsdatascience.com/complete-guide-of-activation-functions-34076e95d044

The multilayer perceptron

- A one-way computational chain
- First, we take the input $\mathbf{x}$ and process it:

  $$\mathbf{h}_1 = g_1(\mathbf{W_1}\mathbf{x} + \mathbf{b_1})$$

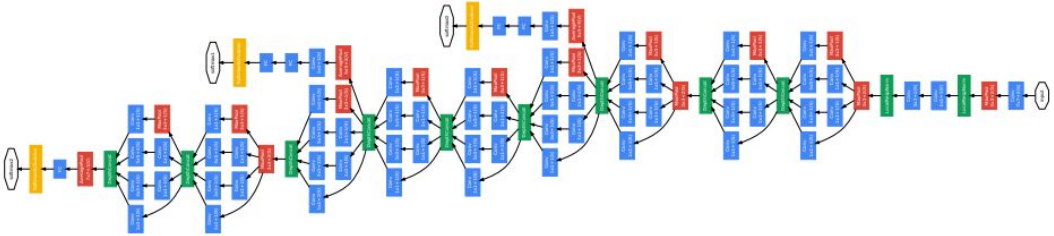
- This gives a first *hidden* representation. Next layer:

  $$\mathbf{h}_2 = g_2(\mathbf{W_2}\mathbf{h_1} + \mathbf{b_2})$$

- \- and so on

Universal approximation

- Think of MLPs as a big function block with many free parameters
- Even with a single hidden layer, *any function* can be represented
  - \- as long as we use a non-linearity
- In practice, single-layer nets may not be trained well to a task
- Instead, we *go deep* and reduce the number of neurons per layer



Szegedy et al., 2015

Recall maximum likelihood

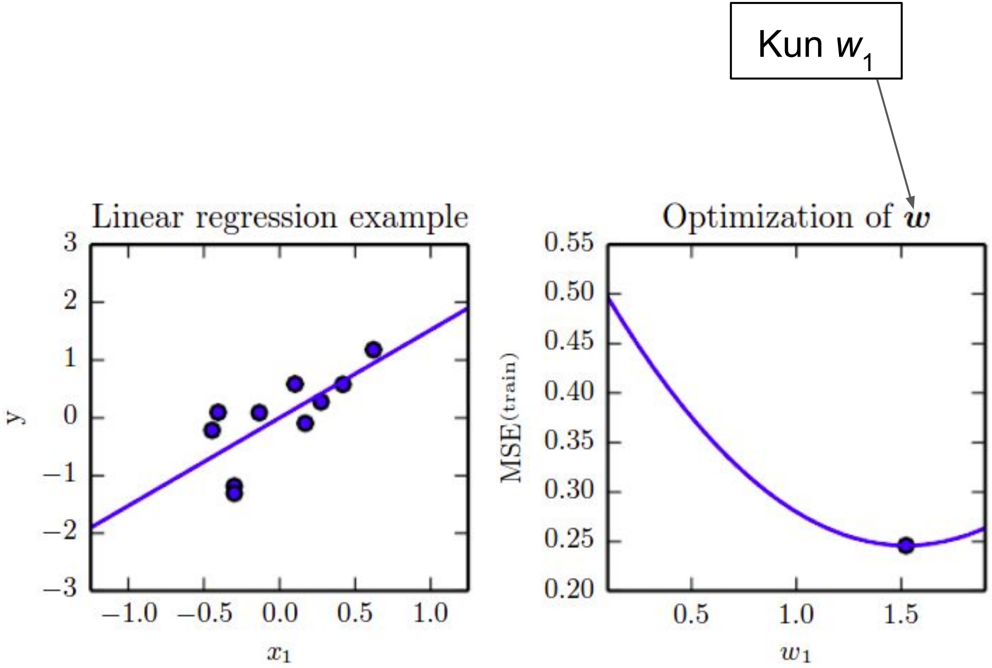
- We setup a (neural) model *p*, which tries to predict an output (label) from an input (e.g. image)
- The ML estimate of its parameters is done using a *training set*:
  $W_{\mathrm{ML}} = \arg\max E_{\mathbf{x}\sim\hat{p}_{\text{data}}} \log p(y|\mathbf{x})$
- where the expectation is simply the mean over the *m* training examples

Generalization error of $W_{\text{ML}} = \arg\max E_{\mathbf{x}\sim\hat{p}_{\text{data}}} \log p(y|\mathbf{x})$

- The problem
  - We do not have access to the full population
  - Instead, we get a limited sample, the *empirical distribution* $\hat{p}_{\text{data}}$
- Now we want our model *p* to follow this empirical distribution - sort of
  - We actually want our model to predict future *test* cases
  - In the end, what we *really* want is high test classification accuracy, but we'll have to do with something we can optimize upon, cross-entropy on the training set

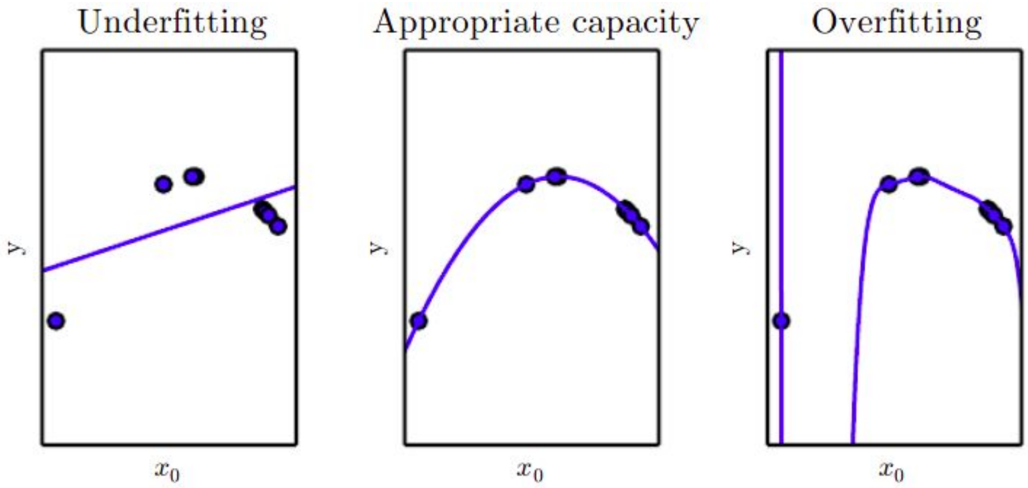
Over- and underfitting

- Take an example
- Best fit line
- Best value of *w* with associated losses



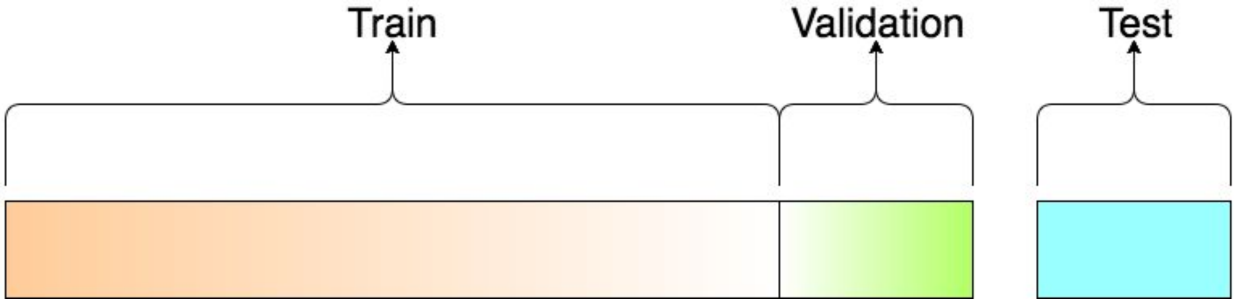
Over- and underfitting

- Another example (training points are sampled on a 2nd degree polynomial)
- Linear regression “fails”
- One more parameter is appropriate
- **This also happens for more realistic training data!**



Train-val-test split

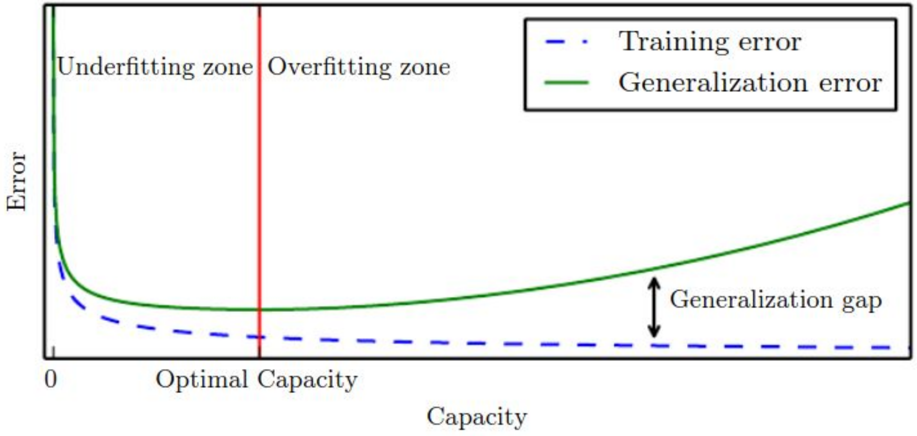
- Usually we get a training set for learning
- The test set is not to be used during learning
  - Sometimes hidden behind an evaluation server
- We can hold out a *validation set* for tuning
  - Also sometimes given by the benchmark
- This allows for design of e.g. network topology



https://towardsdatascience.com/train-validation-and-test-sets-72cb40cba9e7

Network capacity

- The common method to increase capacity of an underfitting model is more weights (more neurons)
- Now we can evaluate this hyperparameter:
  - Full training and evaluation on test set
  - Use the validation set



Exercise

- Introduction to PyTorch
  https://drive.google.com/file/d/1IMC9OJjr-MsgiLTCpAsBNY1JWmXf-cVn/view?usp=sharing
- Multi Layer Perceptron
  https://drive.google.com/file/d/1Go41K2FcH5Ng3sLZKyNm-q55rXCqXJyd/view?usp=sharing

Introduction to PyTorch

- Introduced by FAIR in 2017
- Built with Python as a primary programming language
- Modular and flexible
- Short learning curve & easy to debug

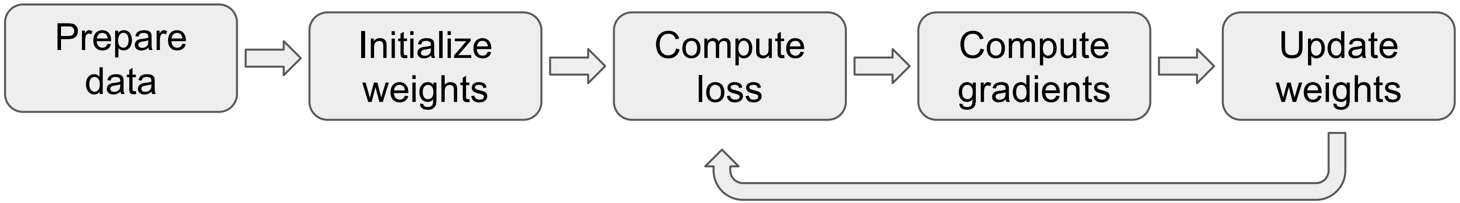
NumPy v/s PyTorch

- Both API’s are very similar
- NumPy arrays -> PyTorch tensors
  - GPU support
  - Automatic differentiation
- Some other differences:
  - NumPy “axis” -> PyTorch “dim”
  - NumPy “reshape” -> PyTorch “view”

Last week - Linear classifiers

- y = ax + b
- Using PyTorch

Optimization loop



torch.nn.module
linear
ReLu
…
custom

Torch.nn.functional.
cross_entropy
mse_loss
…

torch.backward
(computes gradients)

Torch.optim.
SGD
Adam
...

Challenge

- Build a Multi Layer Perceptron (MLP) with one ReLU-activated hidden layer
- Use CIFAR10 dataset
- Use the validation set to determine number of hidden neurons
- Use the test set to get an unbiased estimate of your model's performance on the real data distribution