Neural Networks

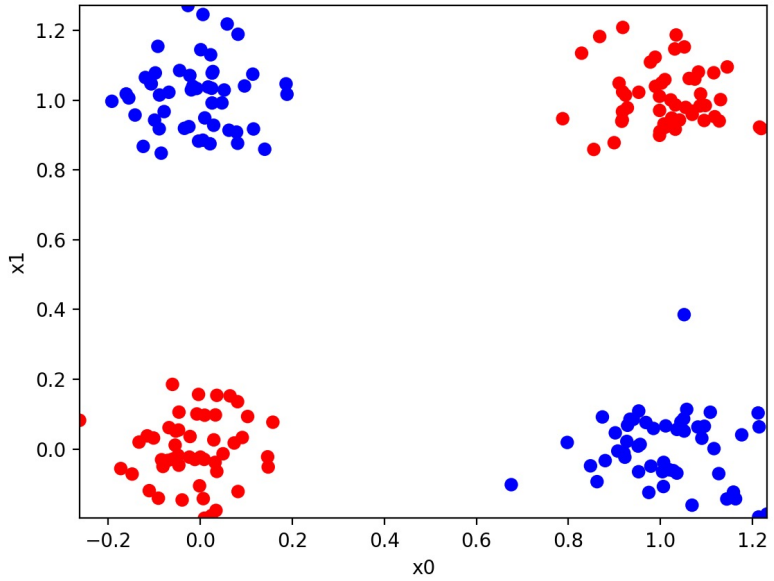
Forest Agostinelli
University of South Carolina

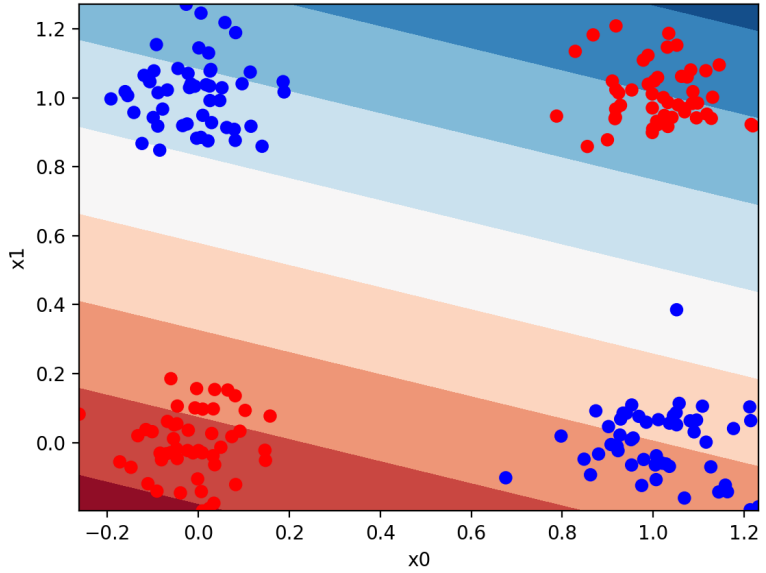
Outline

- The XOR Problem
- Neural networks
- Creating neural network architectures
- Optimization

Linear Model Limitations

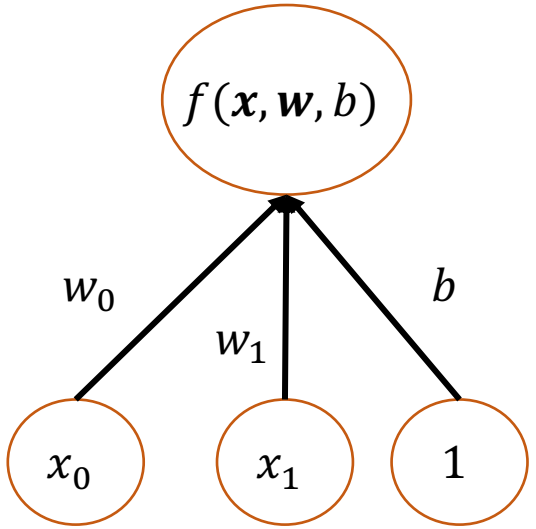
- We saw how linear models cannot capture non-linear relationships between the input and output
- Let's focus on a particular problem called the (noisy) XOR problem
  - If ($x_0 \approx 0$ AND $x_1 \approx 0$) OR ($x_0 \approx 1$ AND $x_1 \approx 1$) then $y = 0$
  - Otherwise ($x_0 \approx 0$ AND $x_1 \approx 1$) OR ($x_0 \approx 1$ AND $x_1 \approx 0$), $y = 1$
- Is capturing this relationship with a linear model possible?

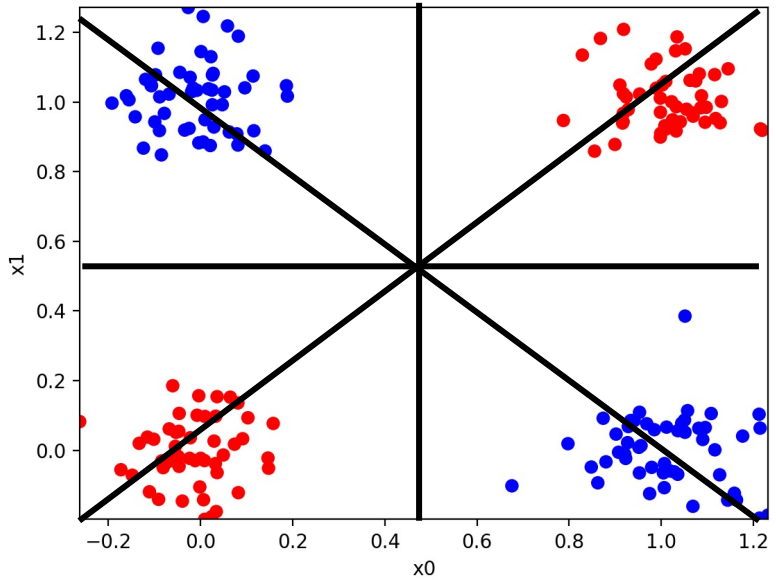




XOR Problem

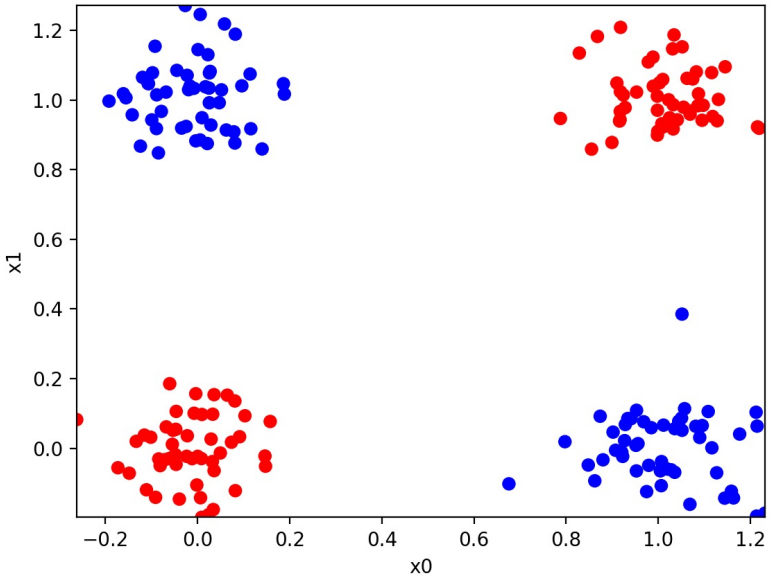
- Let's examine the hypothesis space further to get a better intuition of why this is not possible with a linear model
- No line exists that can separate the two classes





XOR Problem

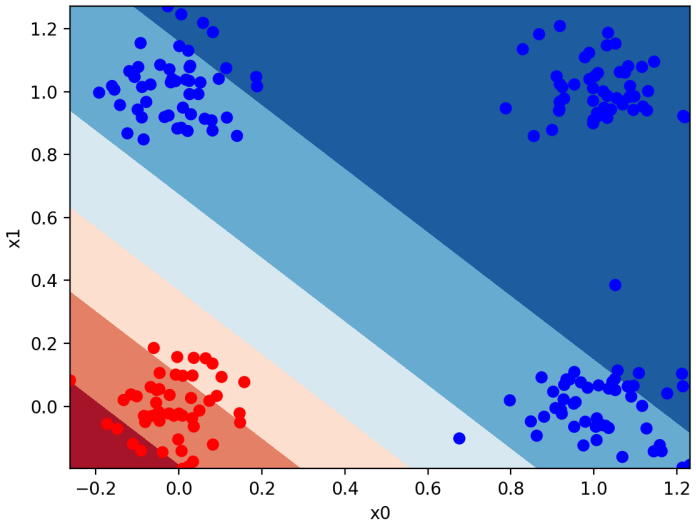
- How can we create a different model whose hypothesis space is able to differentiate between these two classes?
- The underlying relationship is:
  - If ($x_0 \approx 0$ AND $x_1 \approx 0$) OR ($x_0 \approx 1$ AND $x_1 \approx 1$) then $y = 0$
  - Otherwise ($x_0 \approx 0$ AND $x_1 \approx 1$) OR ($x_0 \approx 1$ AND $x_1 \approx 0$), $y = 1$
- Perhaps we can decompose the problem down into sub problems



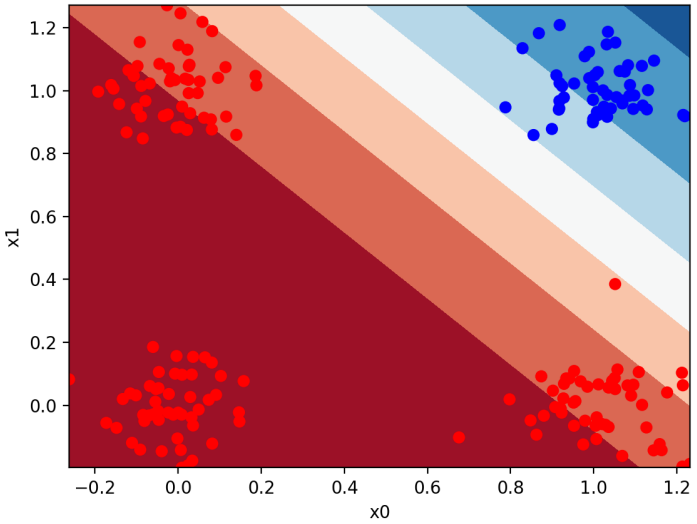
XOR Problem

- We can use a linear classifier on a portion of the data
- How can we build on these two linear classifiers to create another dataset that can be separated linearly?

$(x_0 \approx 1)$ OR $(x_1 \approx 1)$



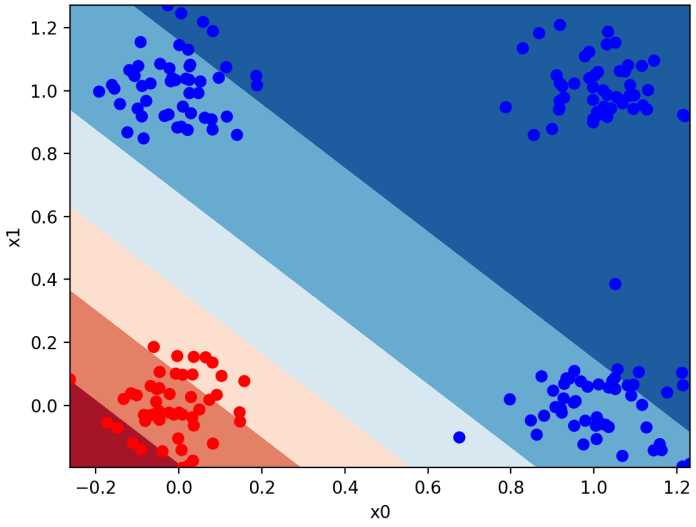
$(x_0 \approx 1)$ AND $(x_1 \approx 1)$



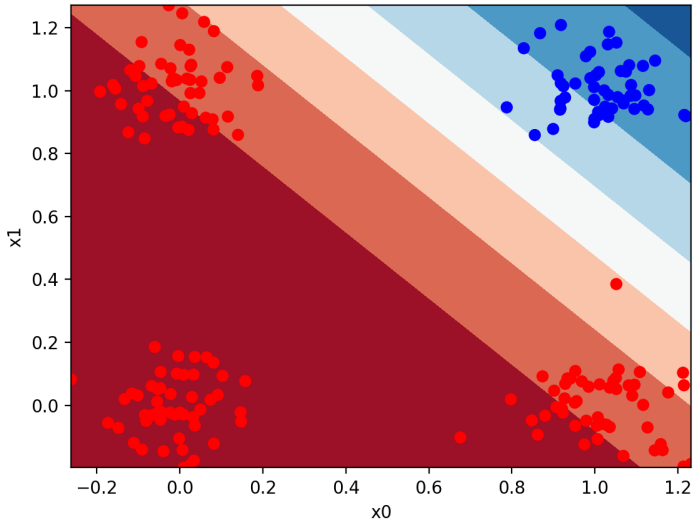
XOR Problem

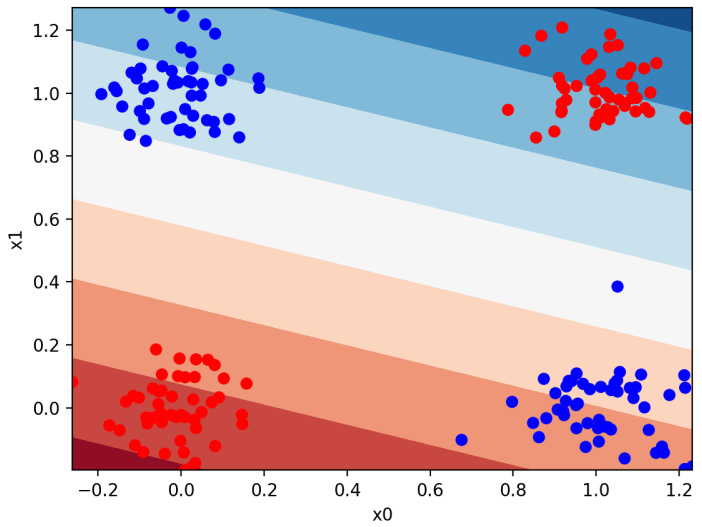
- If $(x_0 \approx 1)$ OR $(x_1 \approx 1)$ is true and $(x_0 \approx 1)$ AND $(x_1 \approx 1)$ is false, then it must be class 1
  - In other words, one of the inputs is 1, but not both
- Otherwise, it must be class 0

$(x_0 \approx 1)$ OR $(x_1 \approx 1)$



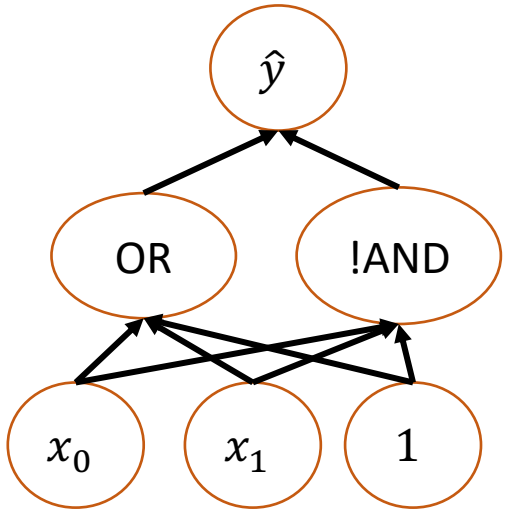
$(x_0 \approx 1)$ AND $(x_1 \approx 1)$

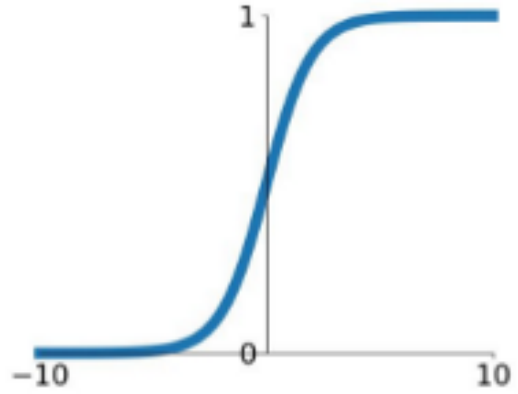




XOR Problem

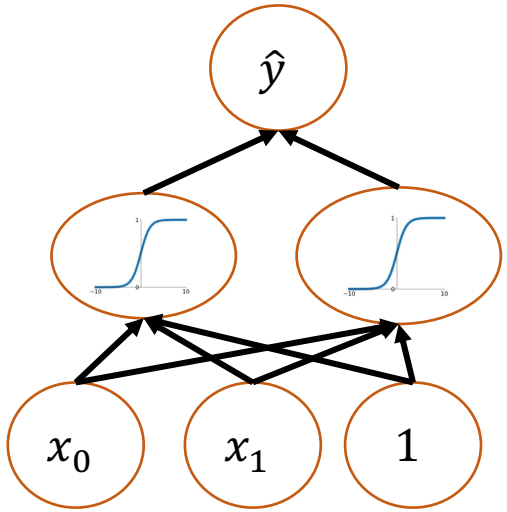
- If $(x_0 \approx 1)$ OR $(x_1 \approx 1)$ is true and $(x_0 \approx 1)$ AND $(x_1 \approx 1)$ is false, then it must be class 1
- Both the OR and !AND portions can be treated as their own classification problem
  - So, we can use a logistic function at the output of each and train it with logistic regression
- We can then train the final classifier with logistic regression
- Can we combine these two steps into one?

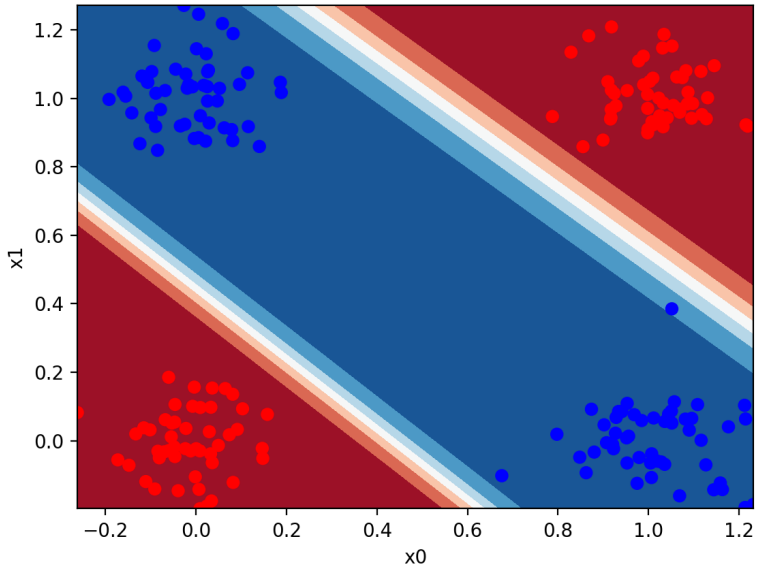




XOR Problem

- Instead of pre-defining what each portion of our model will do, we can define the model and train all the parameters with gradient descent



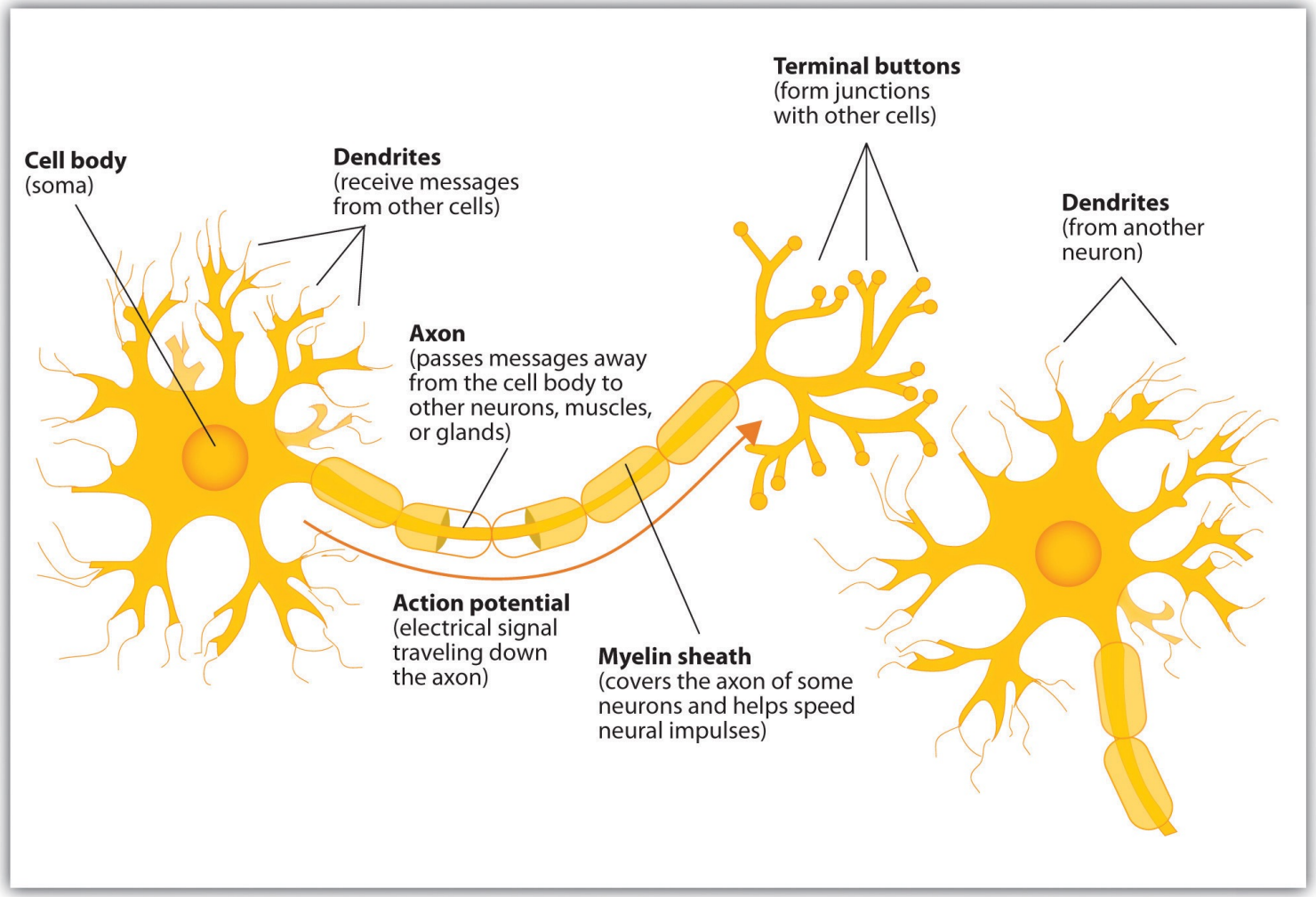


Outline

- The XOR Problem
- Neural networks
- Creating neural network architectures
- Optimization

Neural Networks

- Artificial neural networks from biological neural networks
- Far from an exact model
- The main parallels are
  - Dendrites (inputs)
  - Action potential (activation function)
  - Axon terminals (outputs)

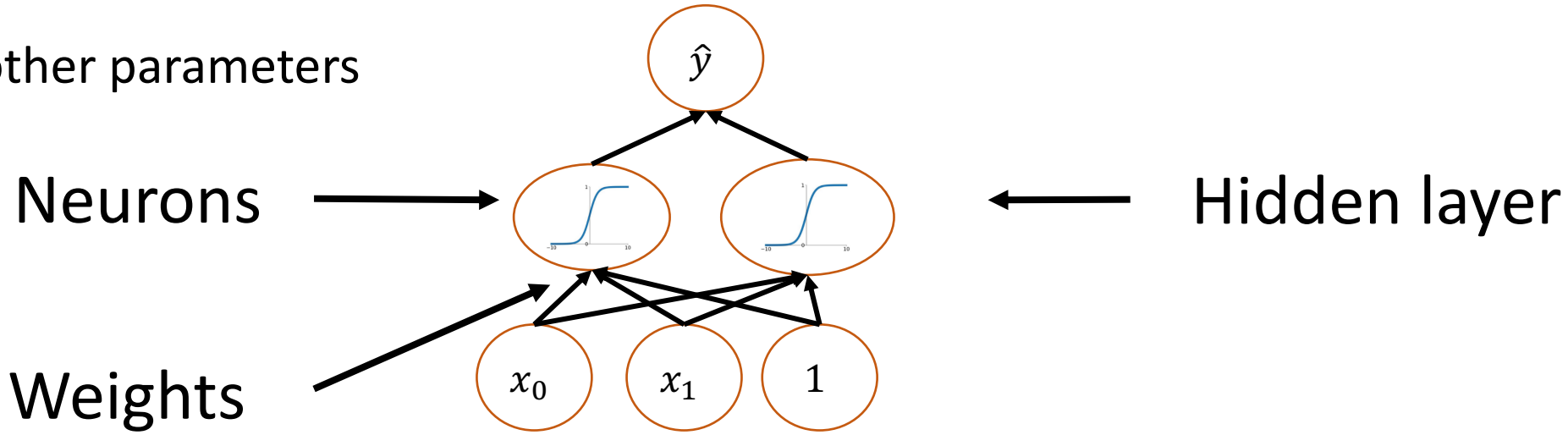


A biological neuron

Jennifer Walinga - https://opentextbc.ca/introductiontopsychology/chapter/3-1-the-neuron-is-the-building-block-of-the-nervous-system/

Neural Networks

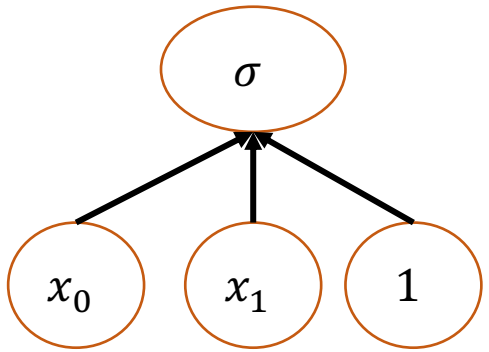
- Neural networks are primarily composed of
  - **Neurons**: Receive input, input determines pre-activation, activation function determines output, output gets sent to other neurons
    - Also referred to as “units”
    - Neurons are often organized into “layers”
  - **Weights:** Form connections between outputs of neurons and inputs of other neurons.
- The parameters of a neural network are often primarily composed of its weights and biases
  - Can have other parameters



Neural Networks: Activation Functions

- Given the input to a neuron, which is often just a linear transformation, the activation function then determines what the output of the neuron will be
- This is often represented by the symbol σ
  - σ is sometimes specific to the logistic function
- What if σ is a linear function?

$$\sigma(w_0 x_0 + w_1 x_1 + b)$$

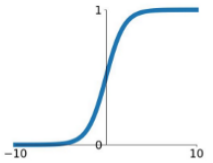


Neural Networks: Activation Functions

- Allow neural network to learn non-linear functions
- Logistic (Sigmoid)
- Rectified Linear Unit (ReLU)
  - Derivative undefined at zero but does not matter in practice
- Activation functions can also be parameterized and learned through gradient descent

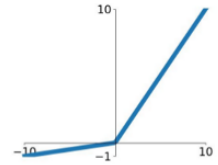
Sigmoid

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



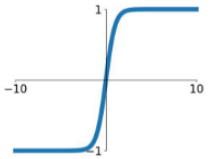
Leaky ReLU

$$\max(0.1x, x)$$



tanh

$$\tanh(x)$$

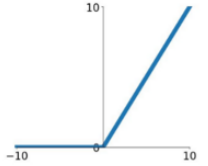


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

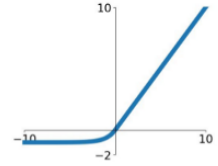
ReLU

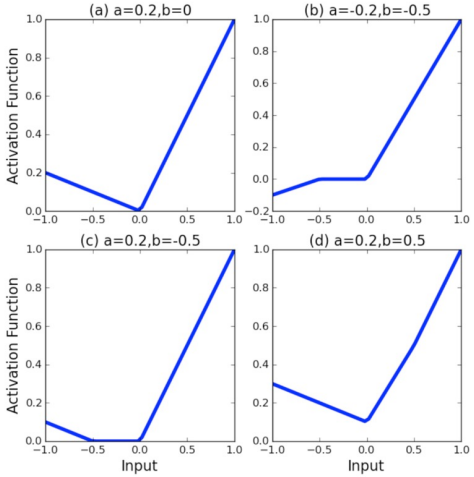
$$\max(0, x)$$



ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$





Adaptive piecewise linear units

https://medium.com/@shrutijadon10104776/survey-on-activation-functions-for-deep-learning-9689331ba092

Agostinelli, F., Hoffman, M., Sadowski, P., & Baldi, P. (2014). Learning activation functions to improve deep neural networks. *arXiv preprint arXiv:1412.6830*.

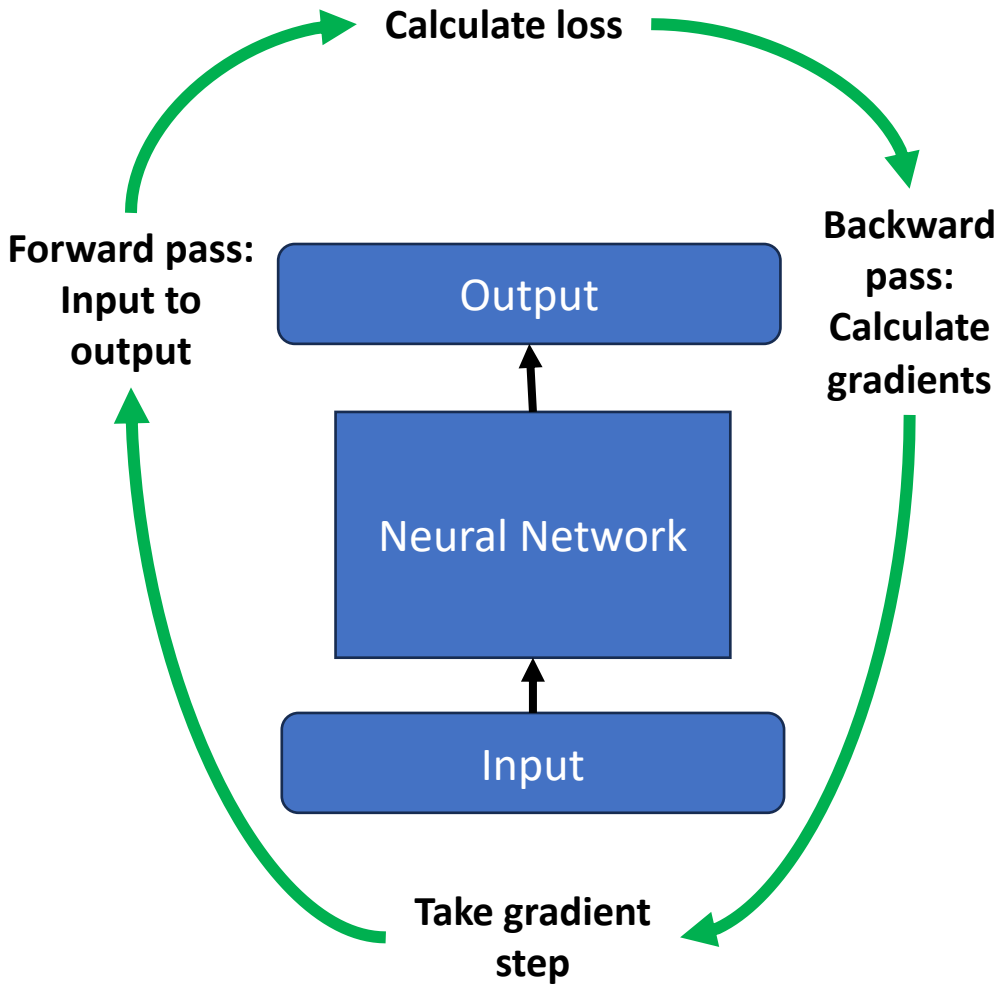
Neural Networks: Universal Function Approximation

- Given enough hidden units, neural networks can approximate any function with arbitrary precision
- Cannot guarantee convergence

Hornik, Kurt, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators." Neural networks 2.5 (1989): 359-366.

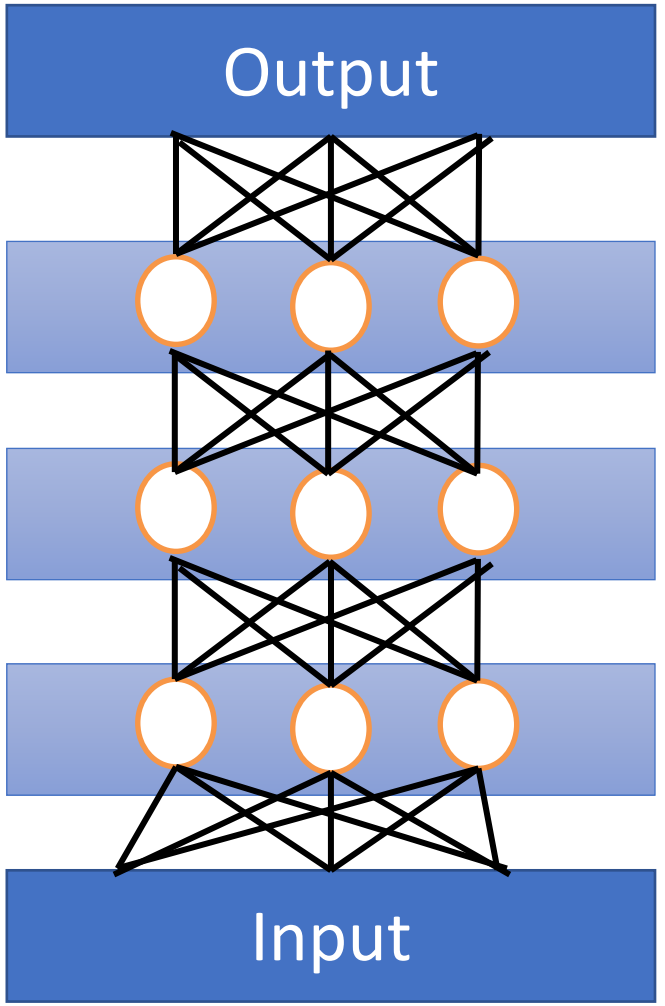
Neural Networks: Backpropagation

- In order to do gradient descent, we need to be able to calculate derivatives for all parameters
- Neural networks can have millions or even billions of parameters
- Backpropagation is an algorithm to calculate the gradient using a single “backward pass”
  - Simply an application of the chain rule



Deep Neural Networks

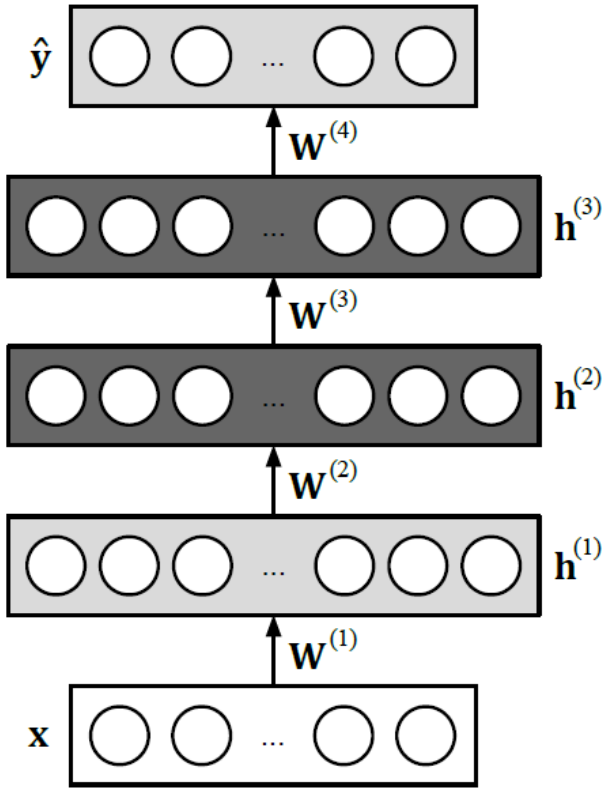
- Stack hidden layers to obtain a deep neural network
- “Deep learning allows computational models that are composed of multiple processing layers to learn representations of data with multiple levels of abstraction.”
- The field of study that focuses on the training and use of deep neural networks is referred to as deep learning

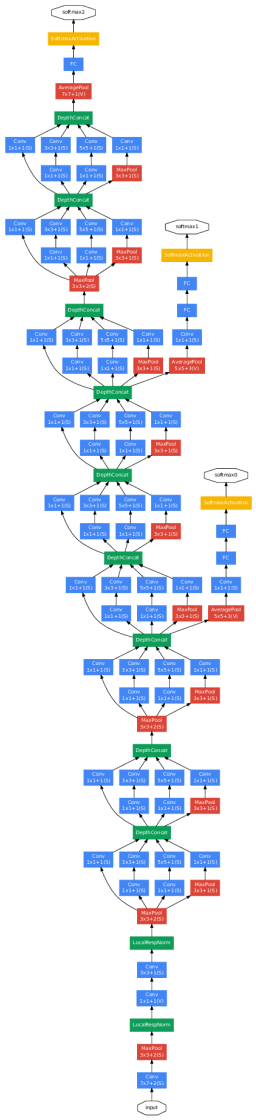


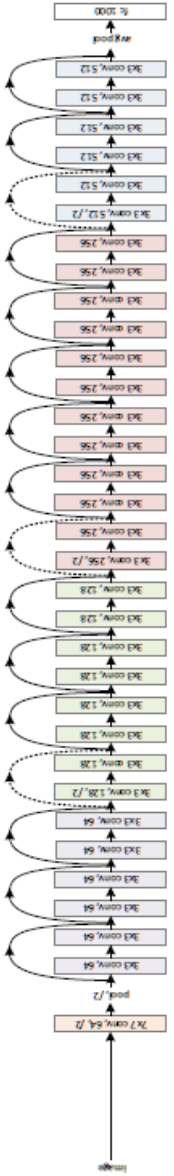
LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. nature, 521(7553), 436.

Deep Neural Networks

- There are a wide variety of ways one can create a deep neural network





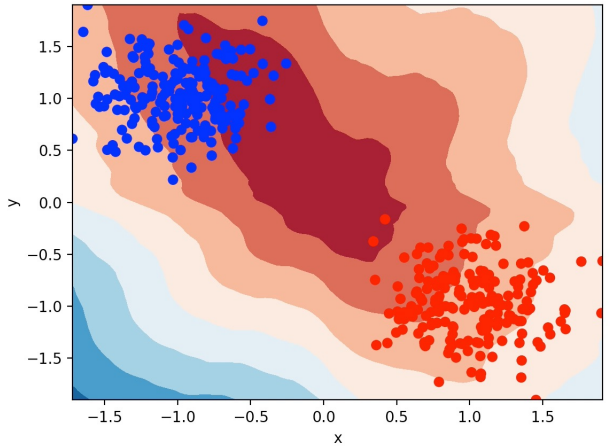


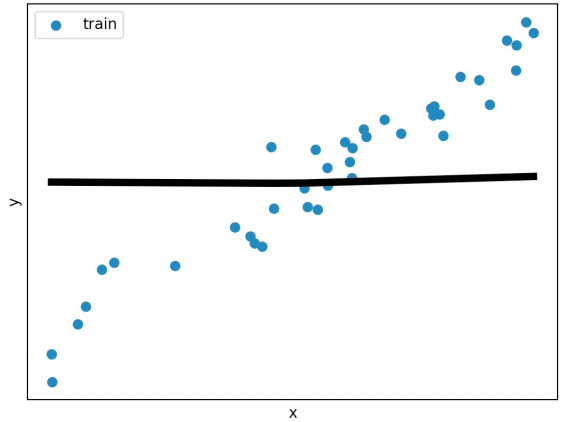
Neural Networks: Regression and Classification

Classification

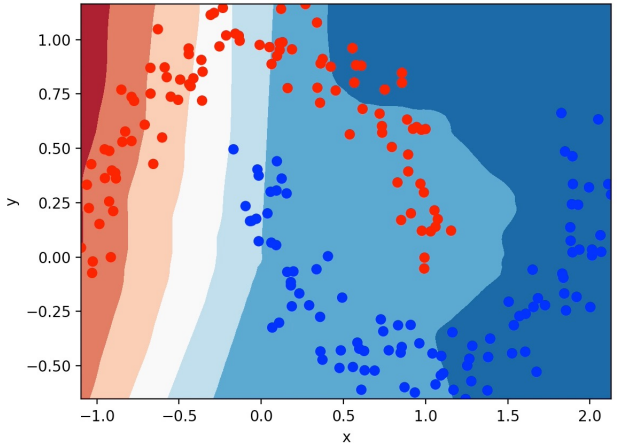
Regression

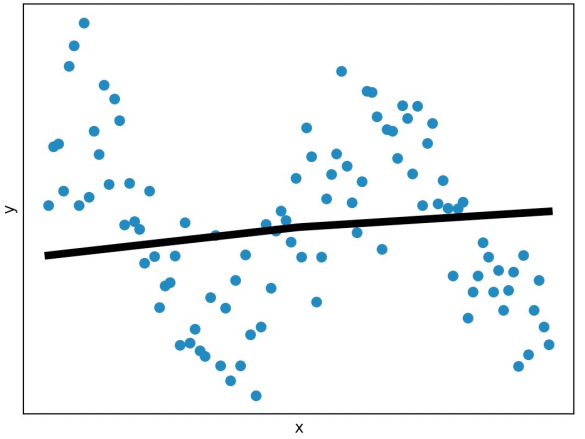
Linear





Non-Linear





Outline

- The XOR Problem
- Neural networks
- Creating neural network architectures
- Optimization

Deep Learning Software

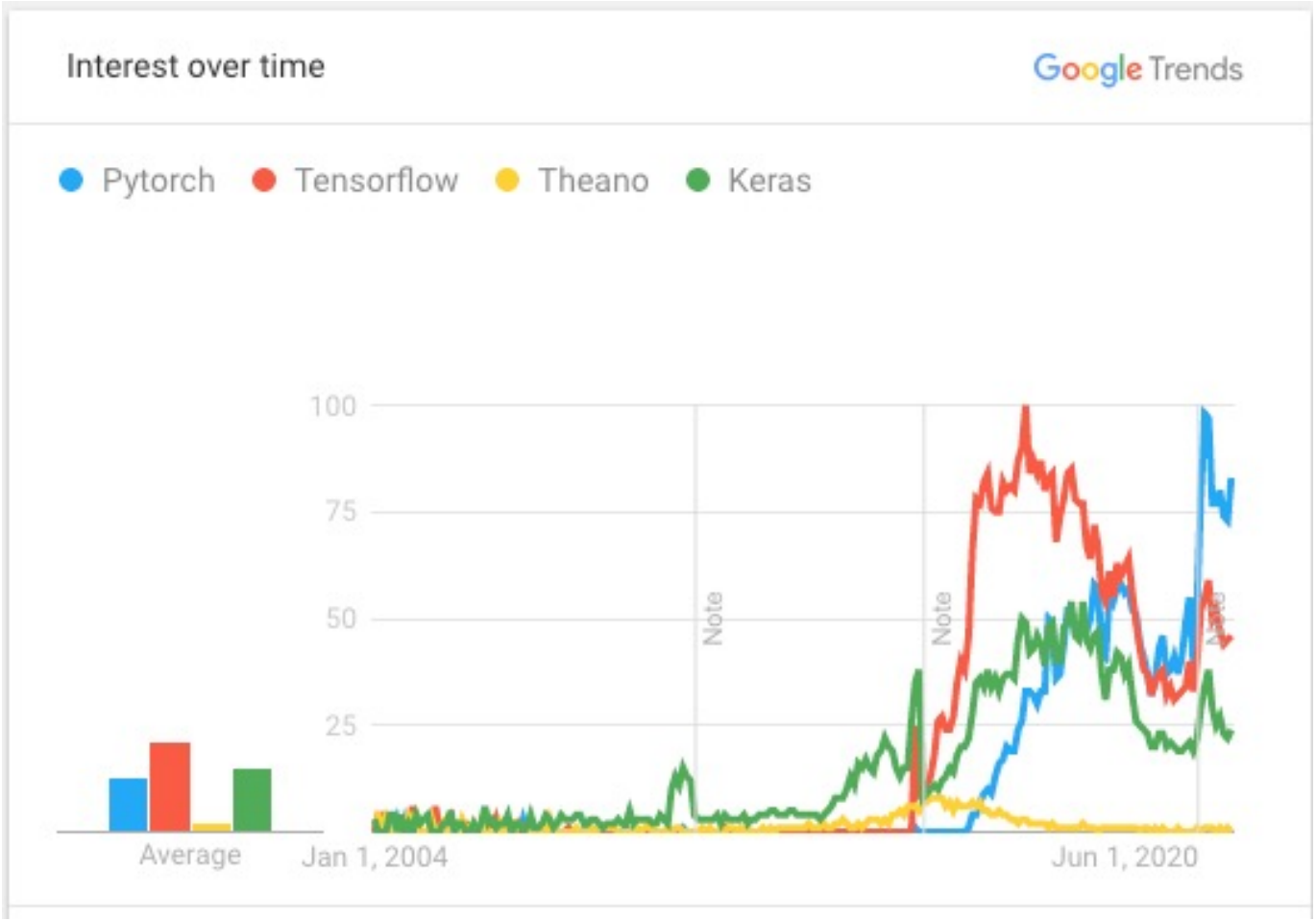
- Building a deep neural network consists of defining a forward pass (obtaining the output) and the backwards pass (backpropagation)
- After backpropagation, the gradient obtained is then used to adjust the parameters
- Furthermore, deep neural networks consist of matrix multiplications which can benefit from parallelization on the simpler, but plentiful, processors of GPUs
- Deep learning software automates some, or all, of this process

Deep Learning Software

- Modern day deep learning software has abstracted away almost all aspects of the backward pass and many aspects of the forward pass
- However, understanding them can be crucial to your research

Deep Learning Software

- Handmade
  - C++/MATLAB/etc.
- Automatic Differentiation
  - Theano
  - Torch
  - Caffe
  - TensorFlow
  - PyTorch



PyTorch: Neural Network Module

- Define parameters
- Define forward pass
- Each entry in the nn.Sequential is its own neural network module
- Applies each module in sequence
- Can use this to easily create useful deep neural networks

```
class NNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(2, 2),
            nn.Sigmoid(),
            nn.Linear(2, 1),
            nn.Sigmoid(),
        )

    def forward(self, x):
        x = self.model(x.float())

        return x
```

Make sure to put comma at the end of each module

PyTorch: Linear Module

- nn.Linear(<input_dimension>, <output_dimension>)
- Creates a linear model, like we saw in the previous lecture
- Can be applied in sequence using activation functions in between
  - nn.Linear(2, 30)
  - nn.Sigmoid()
  - nn.Linear(30, 2)
- Ensure the input dimension of the current one is equal to the output dimension of the previous one

PyTorch: Activation Functions

- Logistic function: nn.Sigmoid()
- Tanh: nn.Tanh()
- Rectified linear: nn.ReLU()
- Exponential linear unit: nn.ELU()

PyTorch: Playing with Architectures

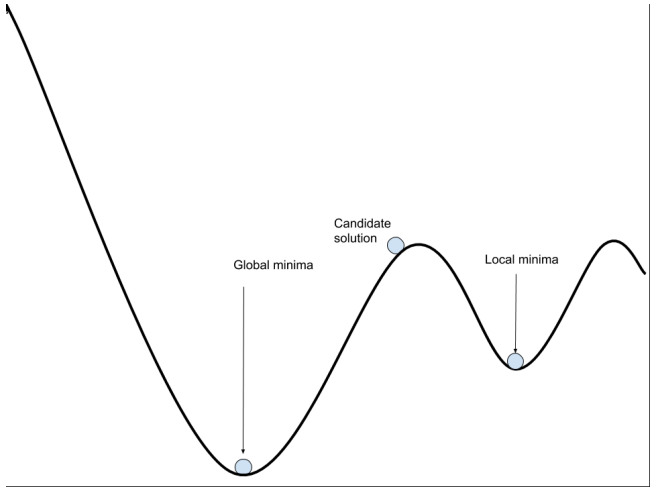
- We can experiment with different neural network architectures for the XOR problem and see which one achieves the lowest loss
- Verify the theory: remove all activation functions in the hidden layers
  - Can it learn a non-linear function?
- Homework: find an architecture that is able to accurately classify the half_moons dataset

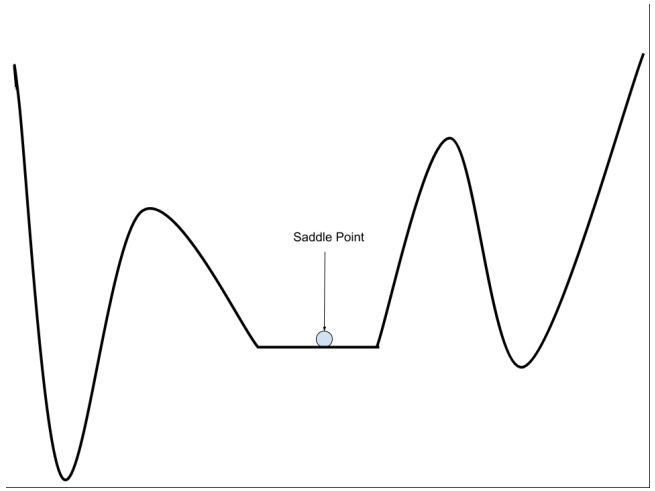
Outline

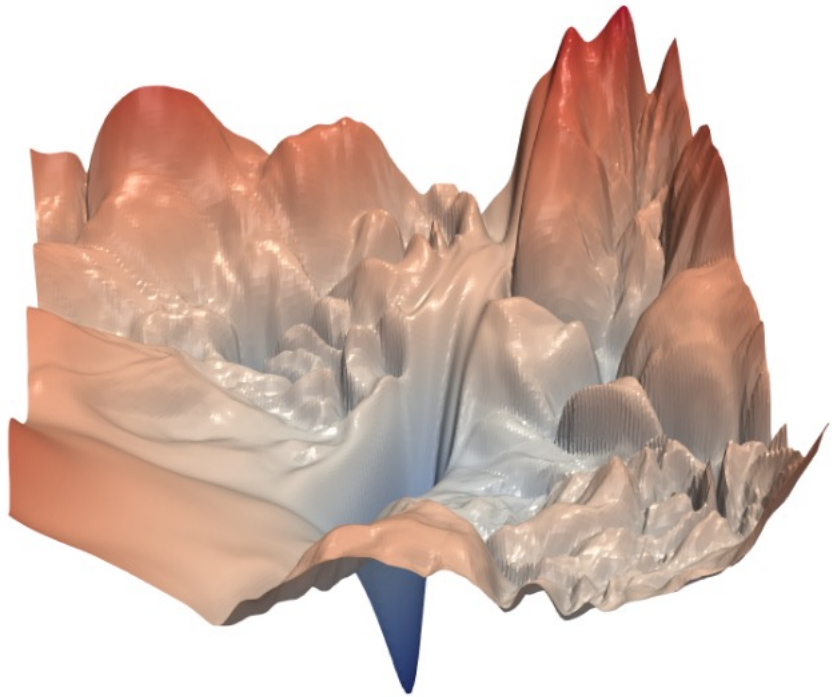
- The XOR Problem
- Neural networks
- Creating neural network architectures
- Optimization

Optimization: Loss Surface

- No longer convex
- Local minima
- Saddle points



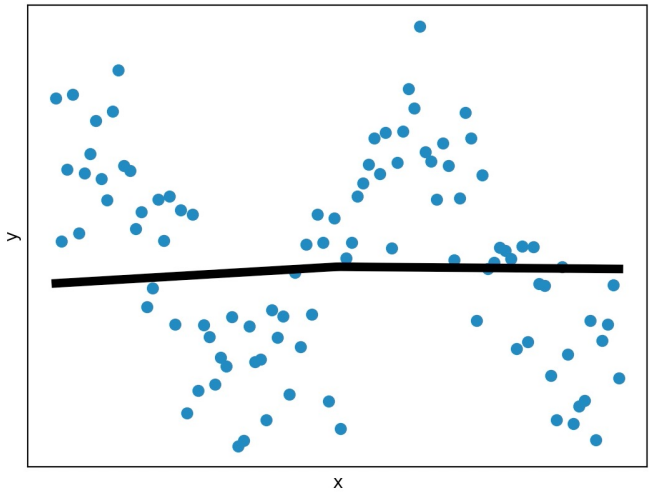




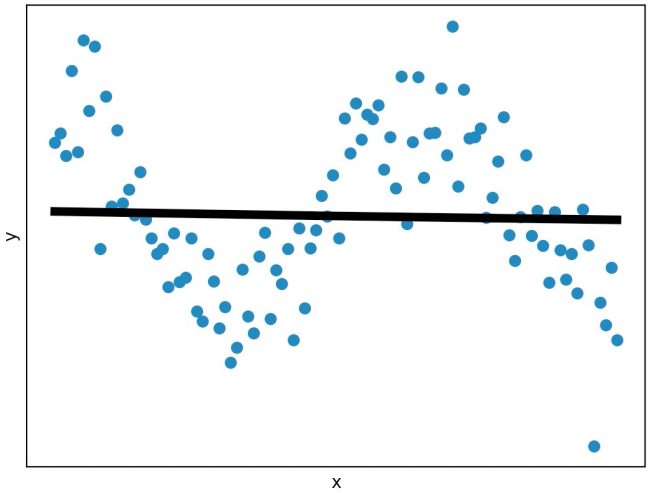
Hyperparameters

- Parameters are learned from the data
- Hyperparameters are set before training
- Learning rate
  - Defines how large the steps will be during gradient descent
  - Usually denoted by α
- Number of neurons
  - How “wide” the neural network is
- Many more!

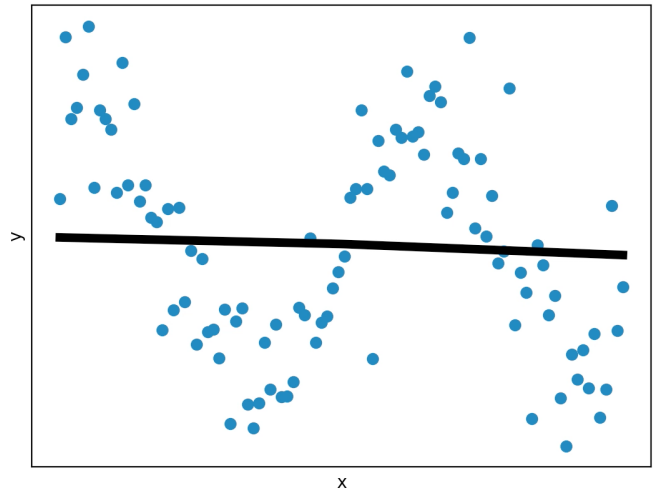
Quick Quiz: What are the Best Hyperparameters?



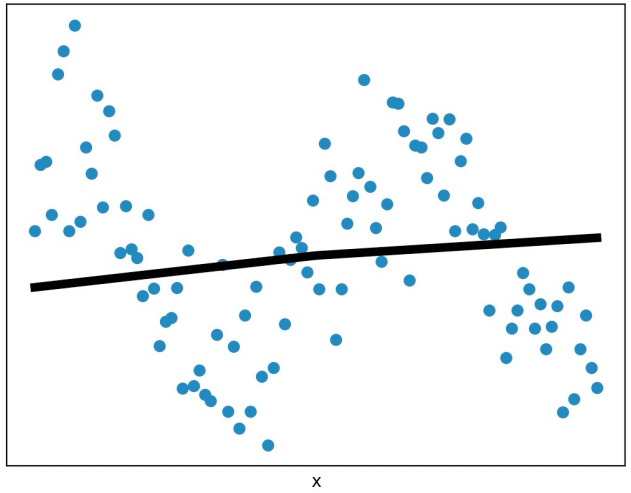
$\alpha = 0.1$, neurons = 100



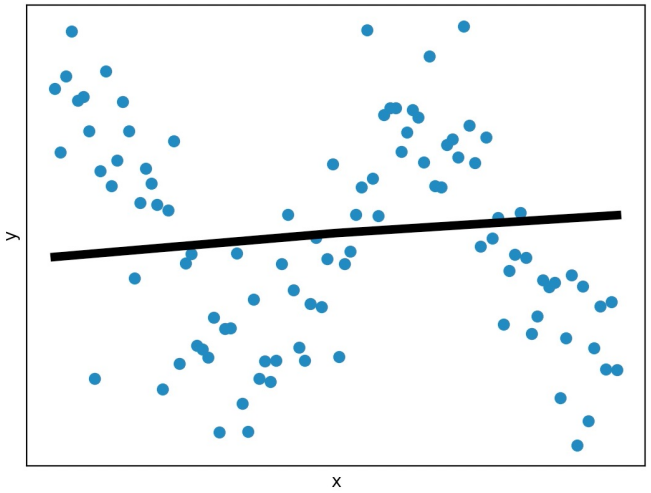
$\alpha = 0.25$, neurons = 100



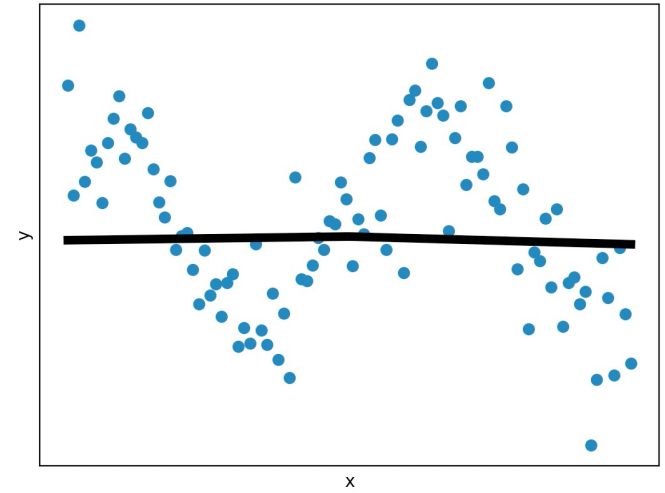
$\alpha = 0.75$, neurons = 100



$\alpha = 0.1$, neurons = 1000



$\alpha = 0.25$, neurons = 1000

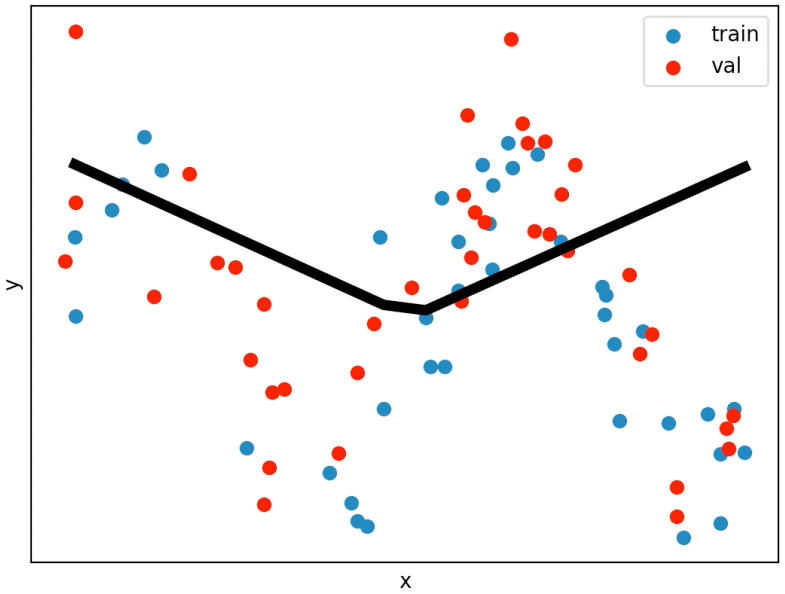


$\alpha = 0.75$, neurons = 1000

Overfitting/Regularization

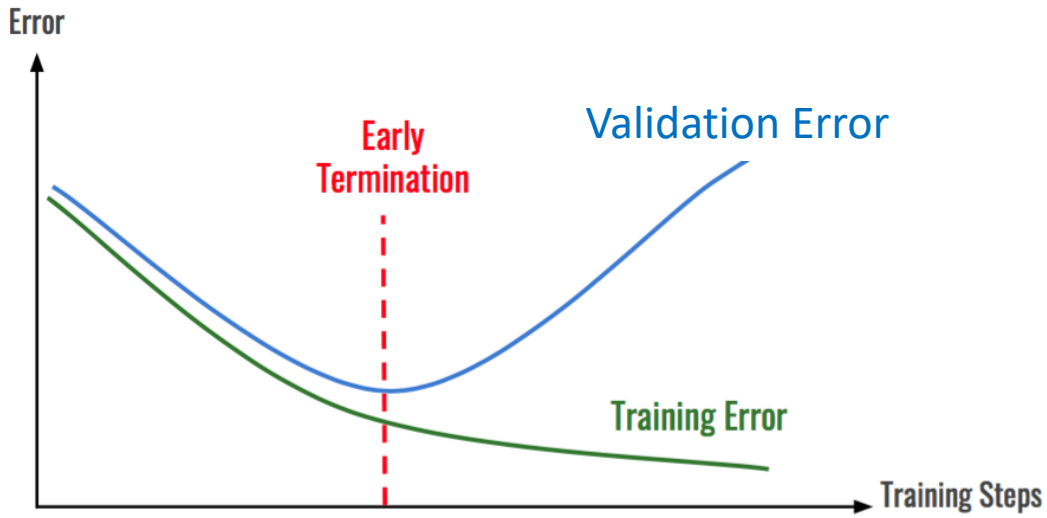
- **Training Dataset**: Used to train neural network
- **Validation Dataset**: Not used to train neural network. Used to determine how well neural network generalizes
- **Test Dataset:** Only for seeing the final performance of neural network. Not used for training or validation.

Loss – Train: 0.042, Val: 0.35



DNN 5 hidden layers and 500 neurons per layer

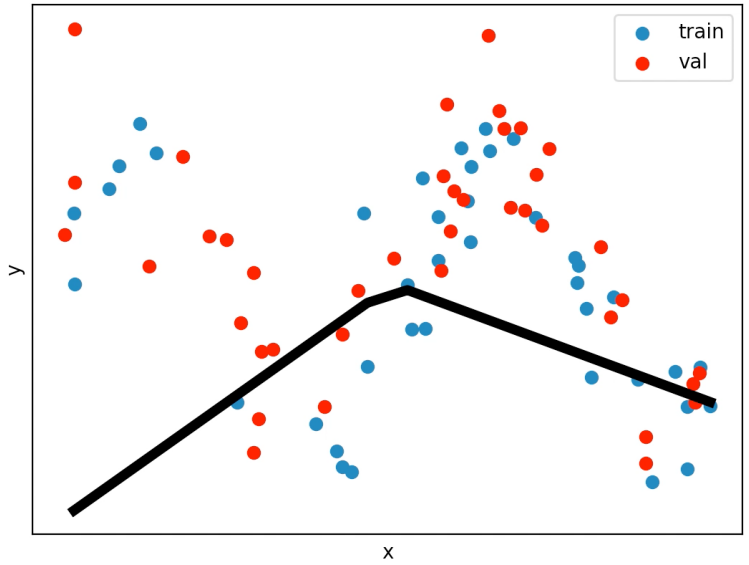
Overfitting/Regularization



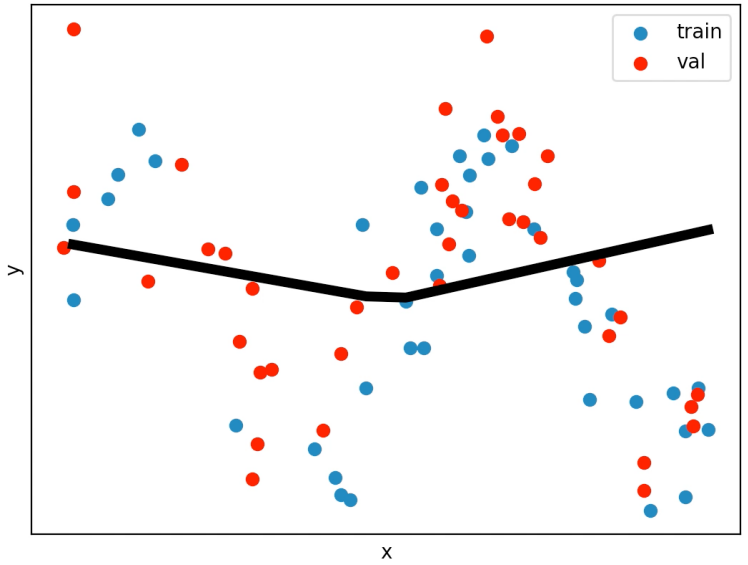
Regularization: Weight Regularization

- Weight regularization
  - $E(\boldsymbol{w}) = \frac{1}{2n}\sum_n ||\boldsymbol{y}_n - \hat{\boldsymbol{y}}_n||_2^2 + \lambda \sum_l \sum_i \sum_j W_{ij}^2$
  - Make the weights less "sensitive" to the input

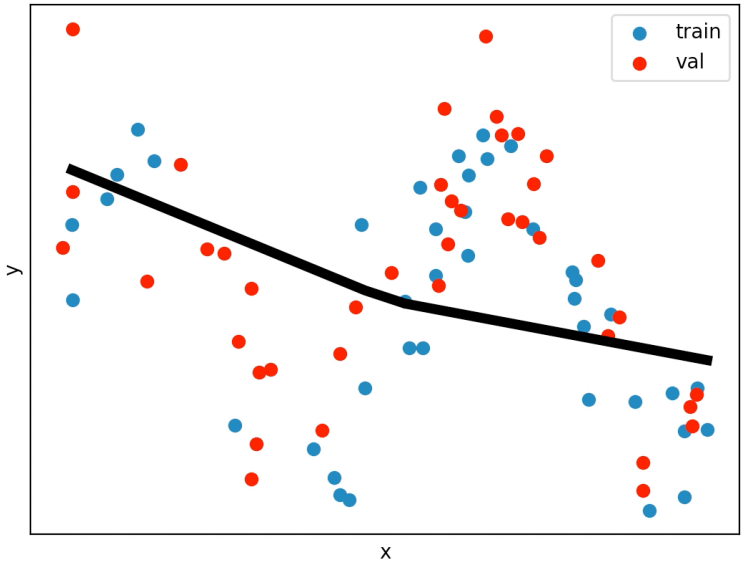
Loss – Train: 0.12, Val: 0.29, $\lambda = 0.01$



Loss – Train: 0.18, Val: 0.26, $\lambda = 0.05$

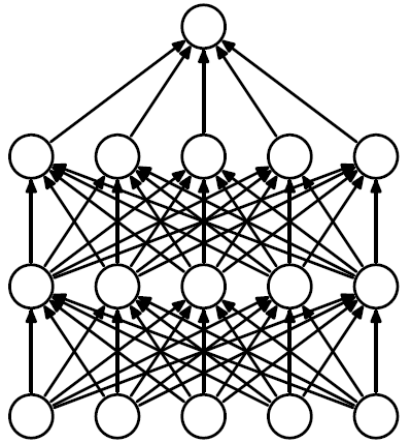


Loss – Train: 0.70, Val: 0.77, $\lambda = 0.2$

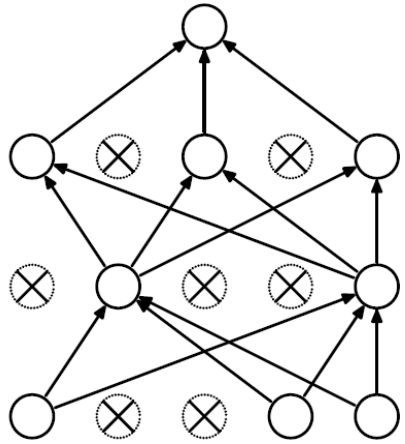


Overfitting/Regularization: Dropout

- Dropout
  - Randomly drop connections between neurons during training

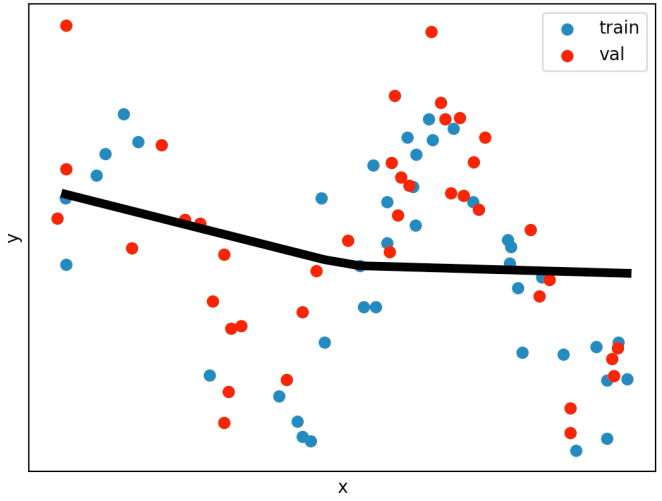


(a) Standard Neural Net

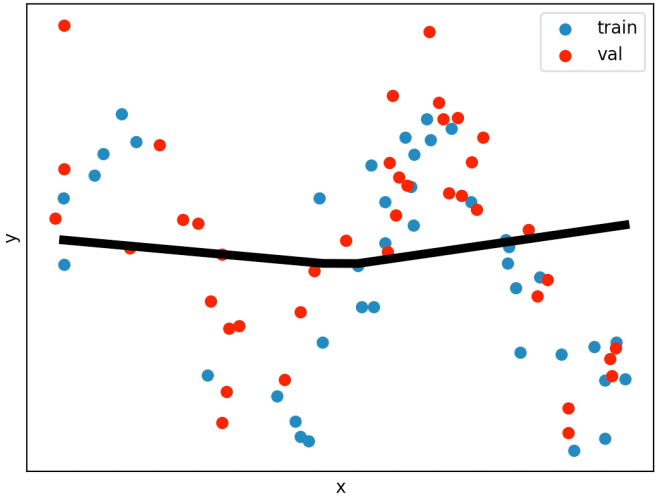


(b) After applying dropout.

Loss – Train: 0.23, Val: 0.25, $p = 0.5$



Loss – Train: 0.70, Val: 0.75, $p = 0.9$



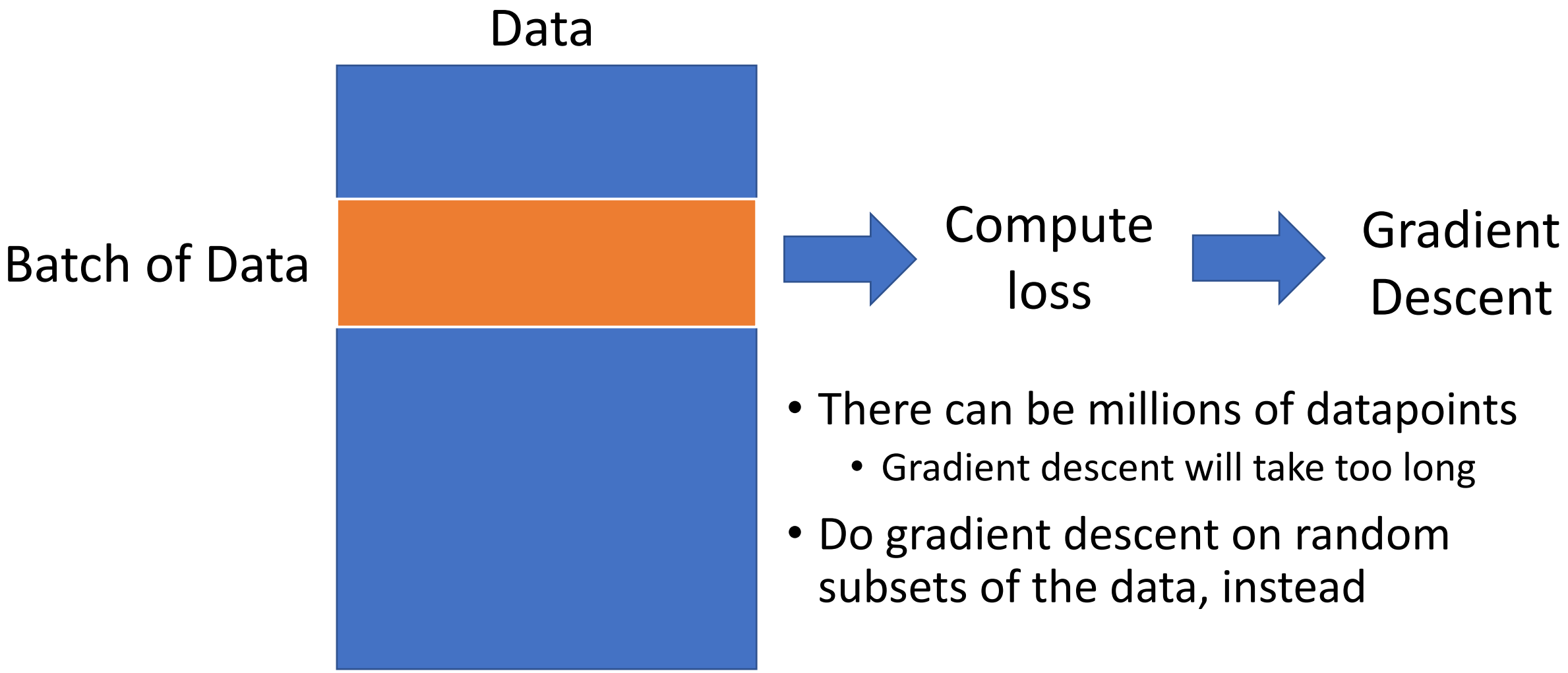
Regularization: Add More Data

- Harder to overfit if there is more data
- Collect more data
- Augment current data
  - Rotations, flipping, translations
  - Adding noise

Optimization: Gradient Based Methods

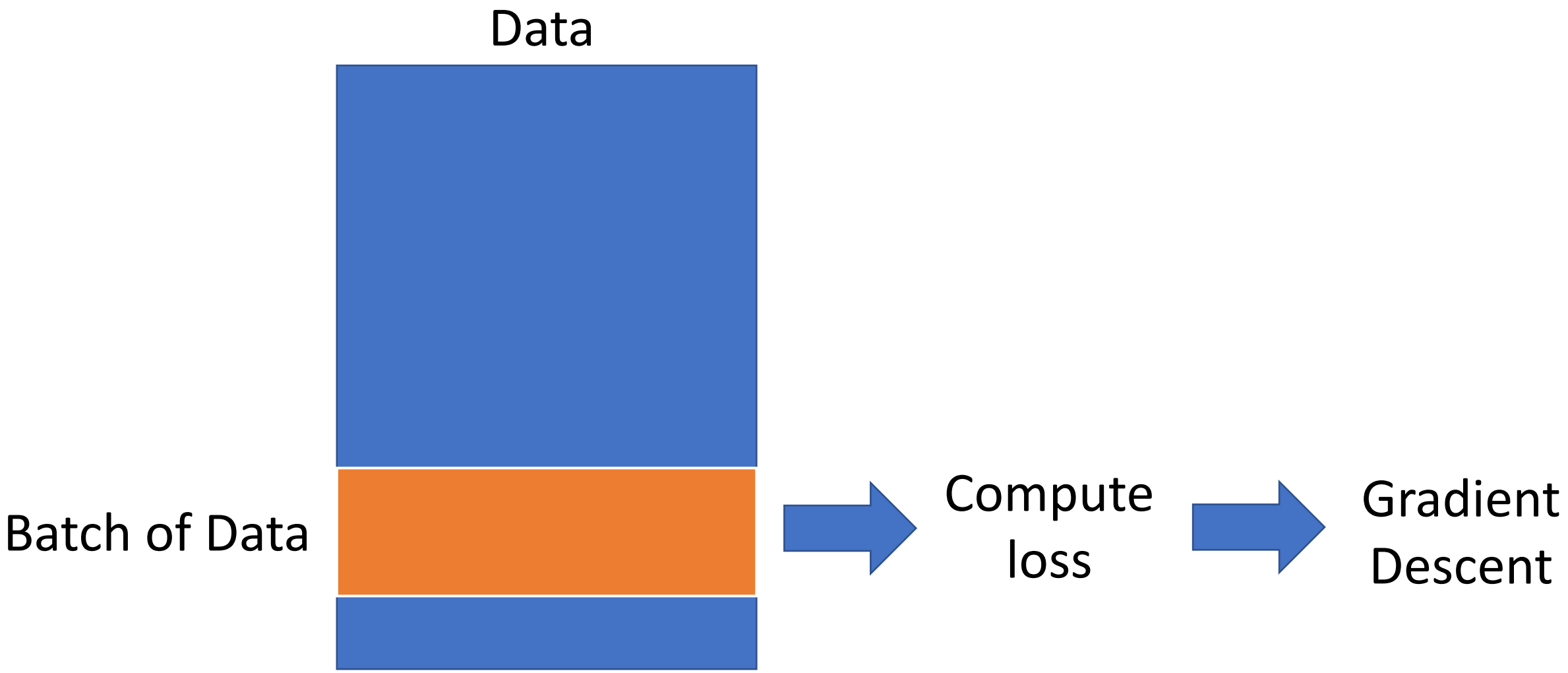
- Vanilla Gradient Descent
  - $\boldsymbol{w} = \boldsymbol{w} - \alpha \nabla_{\mathbf{w}} \mathrm{E}(\mathbf{w})$
- Gradient Descent with Momentum
  - $\boldsymbol{v} = \mu \boldsymbol{v} + \alpha \nabla_{\mathbf{w}} \mathrm{E}(\mathbf{w})$
  - $\boldsymbol{w} = \boldsymbol{w} - \boldsymbol{v}$
- ADAM
  - $\boldsymbol{m} = \beta_1 \boldsymbol{m} + (1 - \beta_1)(\nabla_{\mathbf{w}} \mathrm{E}(\mathbf{w}))^2$ //estimate of the mean of the gradients
  - $\boldsymbol{v} = \beta_2 \boldsymbol{v} + (1 - \beta_2)(\nabla_{\mathbf{w}} \mathrm{E}(\mathbf{w}))^2$ //estimate of the variance of the gradients
  - $\hat{\boldsymbol{m}}$ and $\hat{\boldsymbol{v}}$ are bias corrected estimates of the mean and variance
  - $\boldsymbol{w} = \boldsymbol{w} - \frac{\alpha}{\sqrt{\hat{\boldsymbol{v}}} + \epsilon} \hat{\boldsymbol{m}}$
- Many others: https://ruder.io/optimizing-gradient-descent/

Optimization: Stochastic Gradient Descent

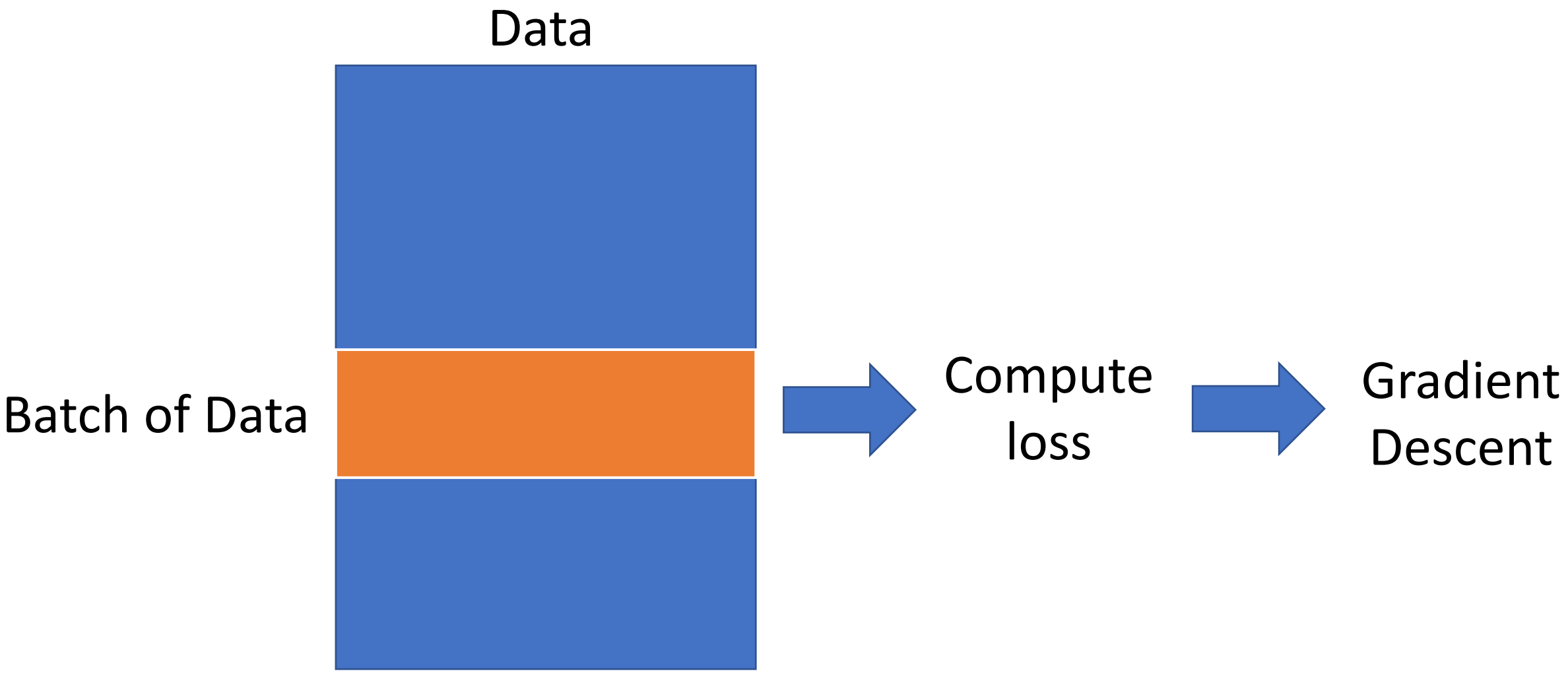


- There can be millions of datapoints
  - Gradient descent will take too long
- Do gradient descent on random subsets of the data, instead

Optimization: Stochastic Gradient Descent

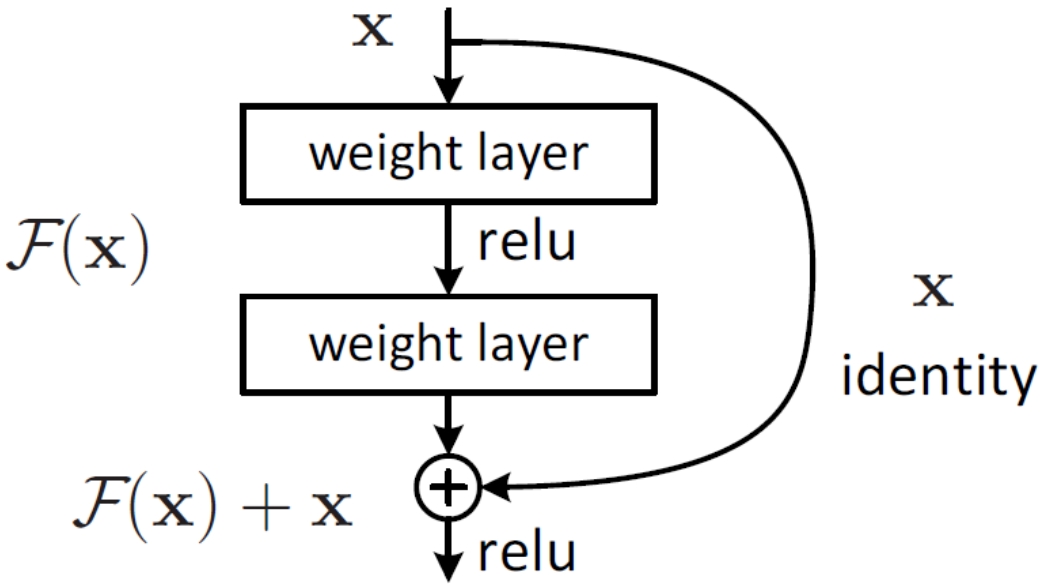


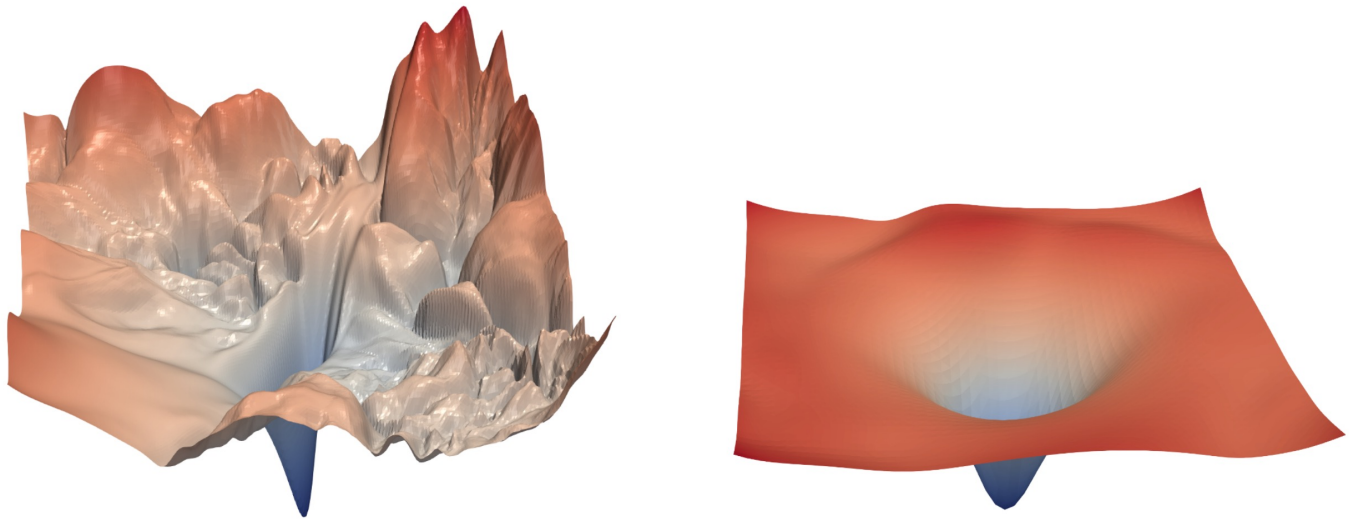
Optimization: Stochastic Gradient Descent



Optimization: Residual Neural Networks

- Training can become more difficult as the number of layers increases
- Adding skip connections allows us to train networks with hundreds of layers





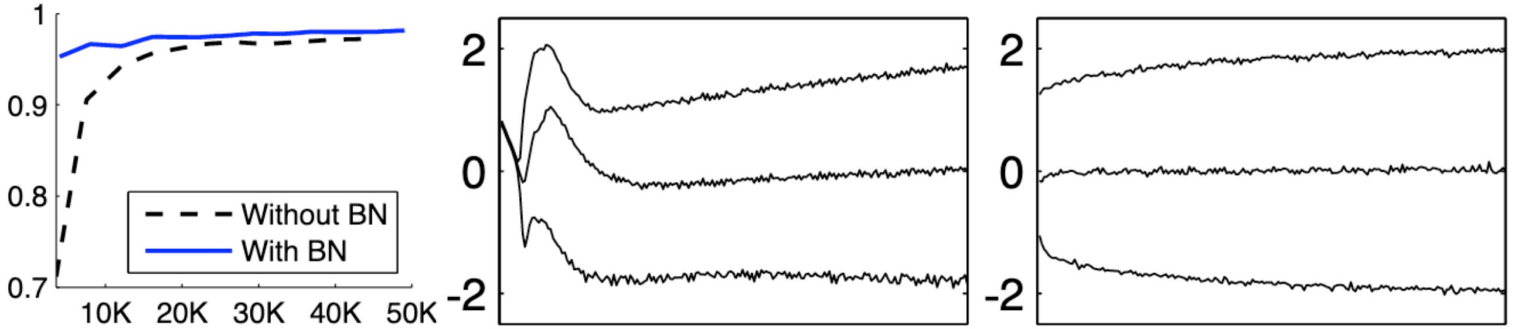
(a) without skip connections (b) with skip connections

Loss surface

Li, Hao, et al. "Visualizing the loss landscape of neural nets." Advances in neural information processing systems 31 (2018).

Optimization: Batch Normalization

- Normalizes the input to the activation function to have a of mean 0 and standard deviation of 1
- Stabilizes training
  - Allows larger learning rates
  - Reduces importance of initialization
- $\boldsymbol{H} = \sigma(BN(\boldsymbol{W}\boldsymbol{X}))$
- Adds some regularization



(a) (b) Without BN (c) With BN

Optimization: Initialization

- The weights of the DNN are randomly initialized
- Initialization can play a large role in optimization
- Xavier/Glorot initialization is fairly common
- Initialization matters less when doing
  - Batch normalization
  - Weight normalization

What to Try?

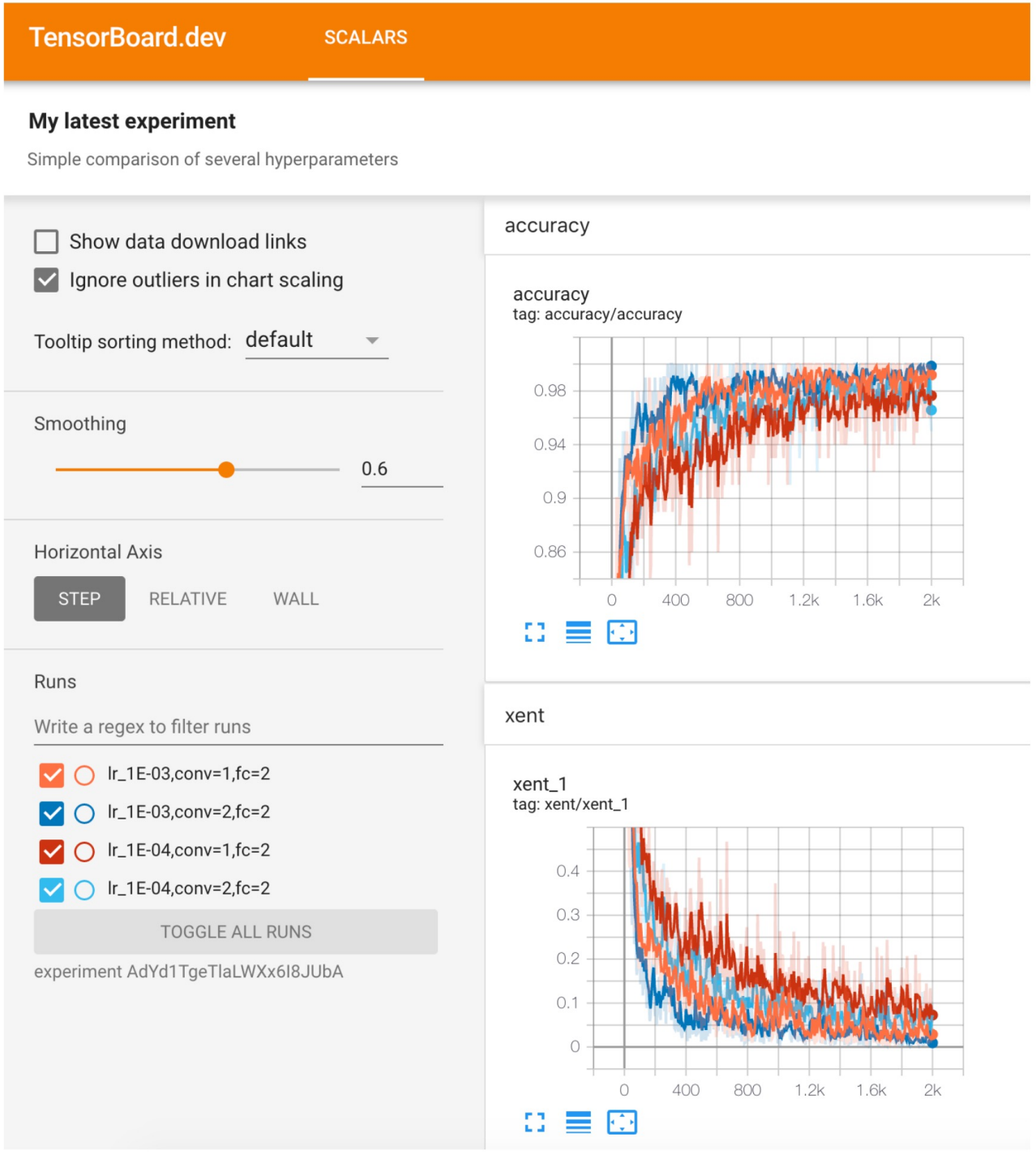
- Activation Function
  - Rectified Linear Units
- Gradient-Based Optimization
  - SGD with momentum
  - ADAM
- Convolution (for structured input like images or sound)
- Batch normalization or Weight normalization
- Residual Networks
- If overfitting?
  - Weight regularization
  - Dropout
    - In higher layers first

Deep Learning/Machine Learning Demos

- https://p.migdal.pl/interactive-machine-learning-list/
- https://cs.stanford.edu/people/karpathy/convnetjs/

TensorBoard

- Developed by TensorFlow
- Useable with PyTorch



PyTorch Tutorial

- PyTorch
  - https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html
- TensorBoard
  - https://pytorch.org/tutorials/recipes/recipes/tensorboard_with_pytorch.html

Relevant Papers

- **Xavier/Glorot Init:** Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010.
- **SGD w/ Momentum:** Sutskever, Ilya, et al. "On the importance of initialization and momentum in deep learning." *International conference on machine learning*. 2013.
- **Imagenet:** Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." *Advances in neural information processing systems*. 2012.
- **ADAM:** Kingma, Diederik P., and Jimmy Ba. "Adam: A method for stochastic optimization." *arXiv preprint arXiv:1412.6980* (2014).
- **Batch Normalization:** Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." *arXiv preprint arXiv:1502.03167* (2015).
- **Residual Networks**: He, Kaiming, et al. "Deep residual learning for image recognition." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016.