Deep Learning Foundations

Winter term 2024/25

Sandipan Sikdar

Leibniz Universität Hannover

Supervised Learning

- We are given a set of training examples of the form

  $\{(x^1, y^1), \dots, (x^n, y^n)\}$

- x^i represents the feature vector for the ith training example

- y^i represents the ground truth

- Goal: Find $f_W: R^d \rightarrow R, f_W(x^i) \approx y^i$

- W: model parameters

Supervised Learning

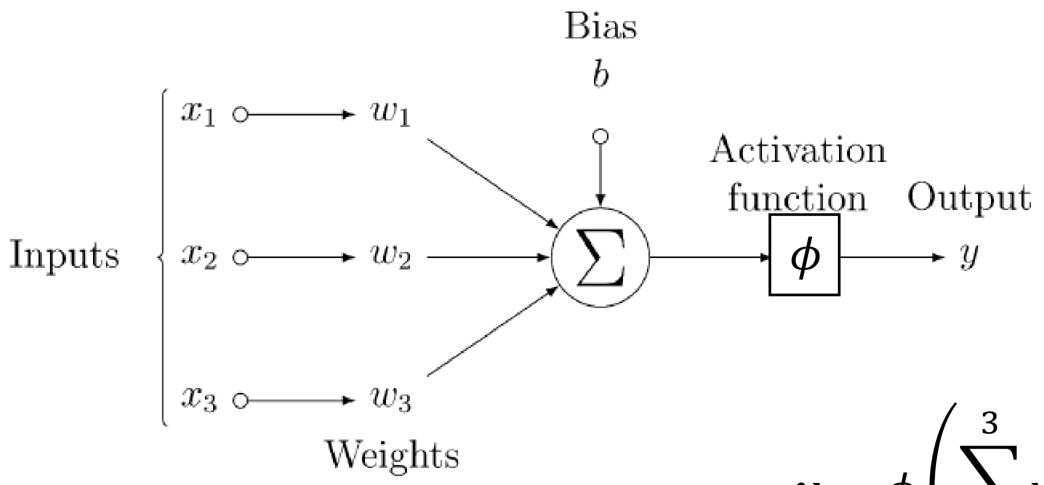
- Goal: Find $f_W: R^d \to R, f_W(x^i) \approx y^i$
- How do we find W?

$$\min_W \frac{1}{n} \sum_{i=1}^{n} l(f_W(x^i), y^i)$$

- l -> loss/cost function
- Emperical risk minimization

Choice of model

- Non-linear functions

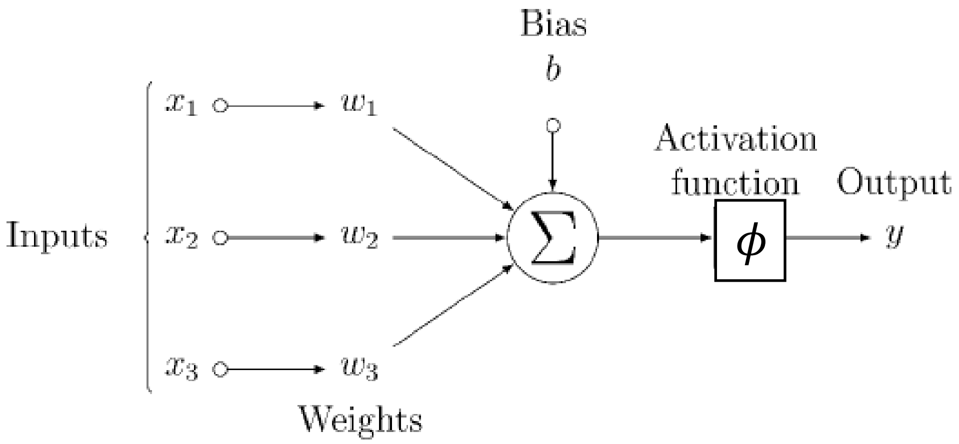


$$y = \phi\left(\sum_{i=1}^{3} w_i x_i + b\right)$$

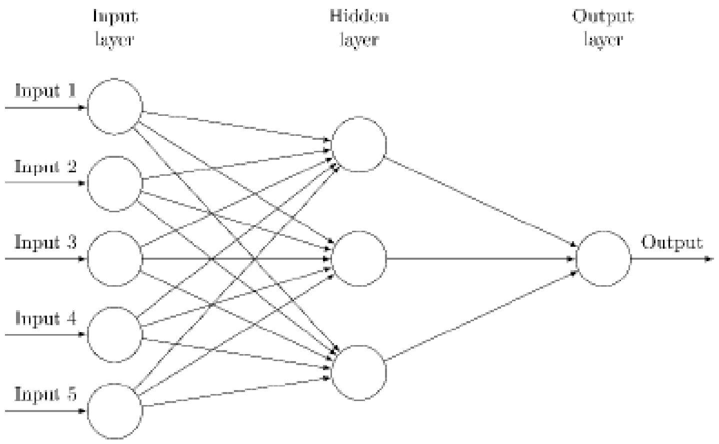
Simplest neural network

Neural Network

- Cascaded layers

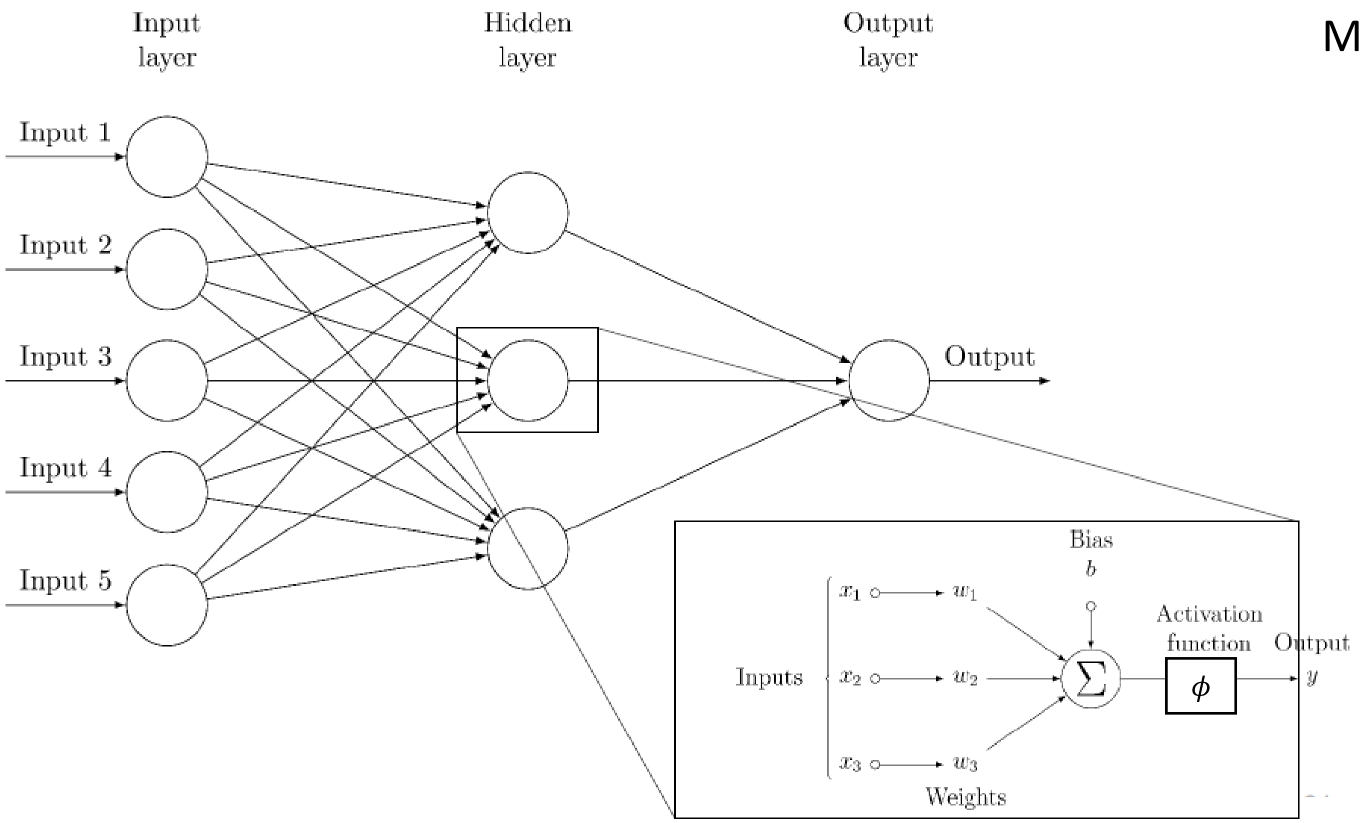


Single Neuron



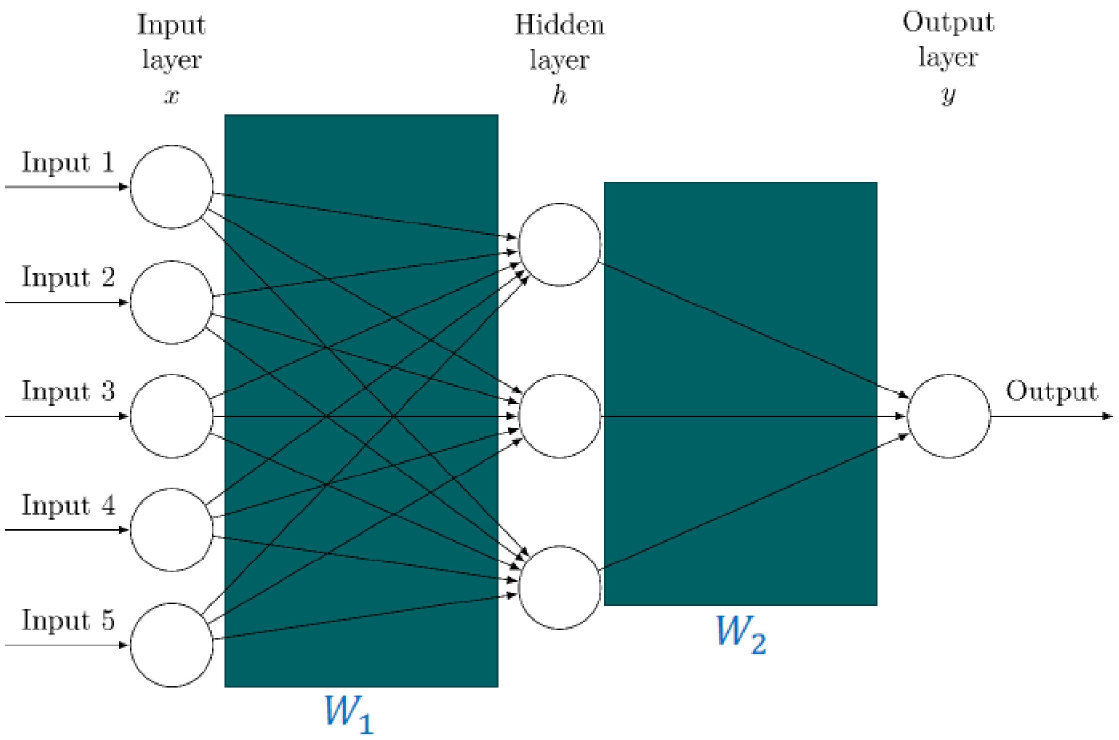
Neural Network

Neural Network



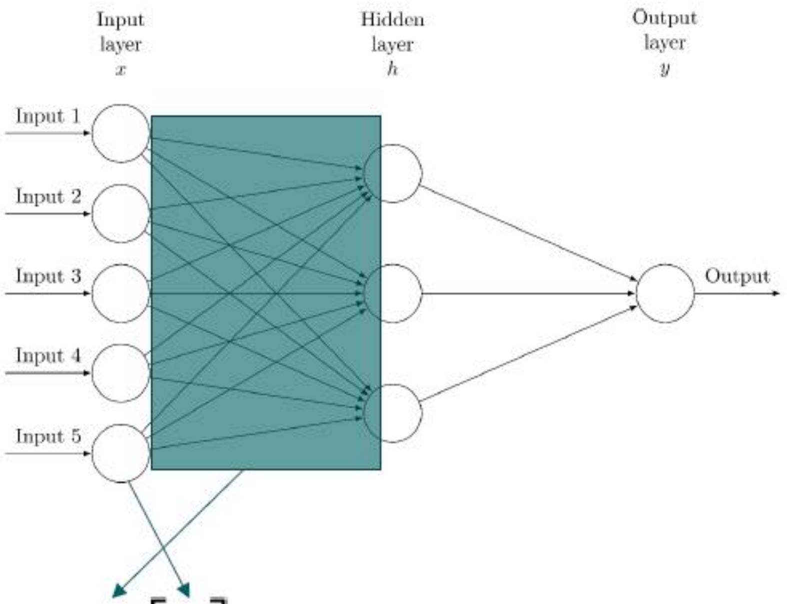
Multi-layer perceptron

Neural Network – Forward Pass



$$h = \phi(W_1 x + b_1)$$
$$y = \phi(W_2 h + b_2)$$

Neural Network – Forward Pass



$$\begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} & w_{15} \\ w_{21} & w_{22} & w_{23} & w_{24} & w_{25} \\ w_{31} & w_{32} & w_{33} & w_{34} & w_{35} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} w_{11}x_1 + w_{12}x_2 + w_{13}x_3 + w_{14}x_4 + w_{15}x_5 + b_1 \\ w_{21}x_1 + w_{22}x_2 + w_{23}x_3 + w_{24}x_4 + w_{25}x_5 + b_2 \\ w_{31}x_1 + w_{32}x_2 + w_{33}x_3 + w_{34}x_4 + w_{35}x_5 + b_3 \end{bmatrix} \xrightarrow[\text{Activation}]{} \begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix}$$

Chained Linear Classifiers

- Remember
  - A linear classifier provides class scores by calculating $y = \phi(Wx + b)$
  - A neural network is a chain of linear classifiers with activation functions

- A neural network is some function like this:

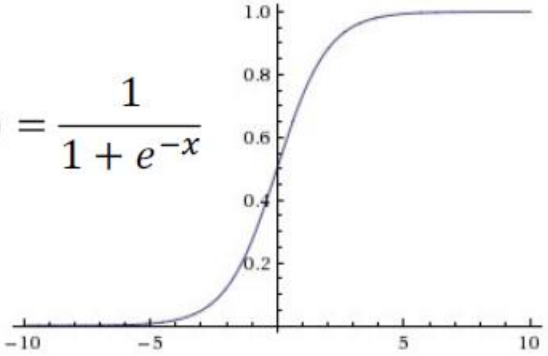
$$y = \phi(W_3\phi(W_2\phi(W_1x + b_1) + b_2) + b_3)$$

- We can chain this as deep as we want

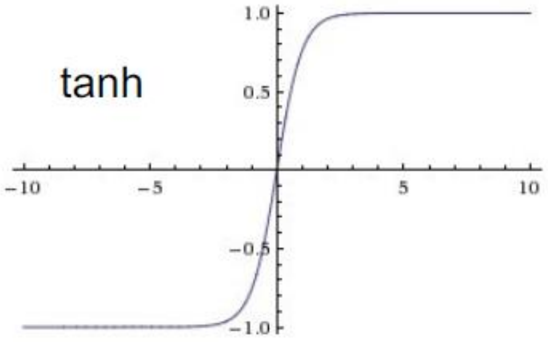
Activation Functions

- Some common activation functions:

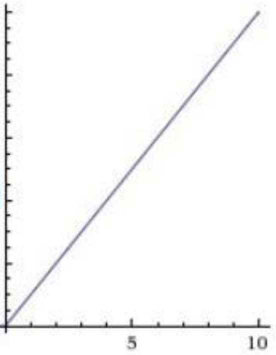
$$\text{sigmoid}(x) = \frac{1}{1+e^{-x}}$$



tanh



ReLU(x) = max(0,x)



Use ReLU!
Works best in practice

Some more:
- Heaviside
- Leaky ReLU
- SeLU
- …

Universal Approximation Theorem

- “A neural network can approximate any continuous function”

Theorem 1. *Let σ be any continuous discriminatory function. Then finite sums of the form*

w_j

$$G(x) = \sum_{j=1}^{N} \alpha_j \sigma(y_j^T x + \theta_j) \qquad (2)$$

are dense in $C(I_n)$. In other words, given any $f \in C(I_n)$ and $\varepsilon > 0$, there is a sum, $G(x)$, of the above form, for which

$$|G(x) - f(x)| < \varepsilon \qquad \textit{for all} \quad x \in I_n.$$

Cybenko G, “Approximation by Superpositions of a Sigmoidal Function”

Training/Optimization

$$\min_W \frac{1}{n}\sum_{i=1}^{n} l(f_W(x^i), y^i)$$

- Use a search algorithm that starts with some initial guess and repeatedly changes W such that the loss gets smaller and smaller until we reach a point where the loss is minimized
- Gradient descent algorithm

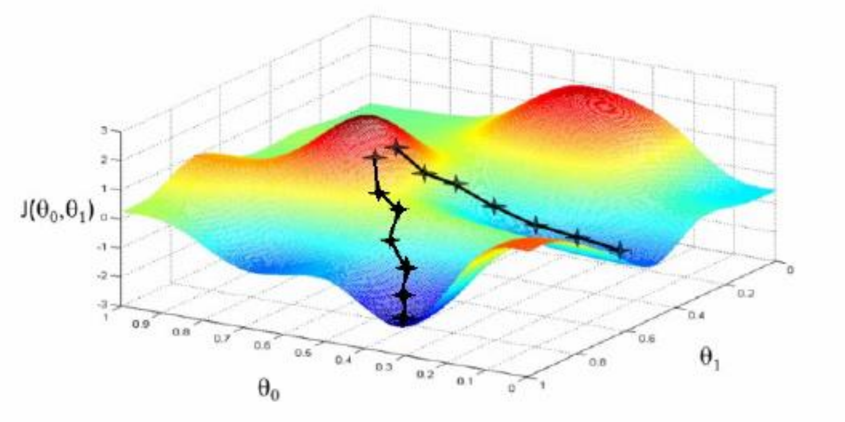
Training/Optimization

$$L(W) = \frac{1}{n}\sum_{i=1}^{n} l(f_W(x^i), y^i)$$

- Gradient descent algorithm

$$W_j = W_j - \underbrace{\alpha}_{\text{Learning rate}} \frac{\partial}{\partial W_j} L(W)$$

Gradient Descent



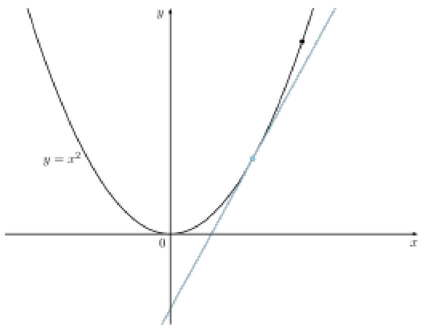
http://blog.datumbox.com/wp-content/uploads/2013/10/gradient-descent.png

Gradient Descent

Gradient Descent

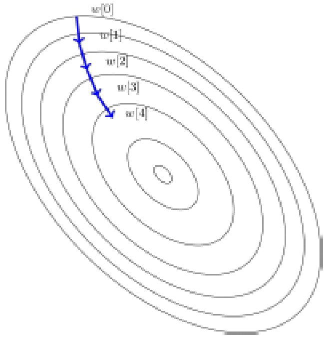
Gradient

= The vector of all partial derivatives of a function



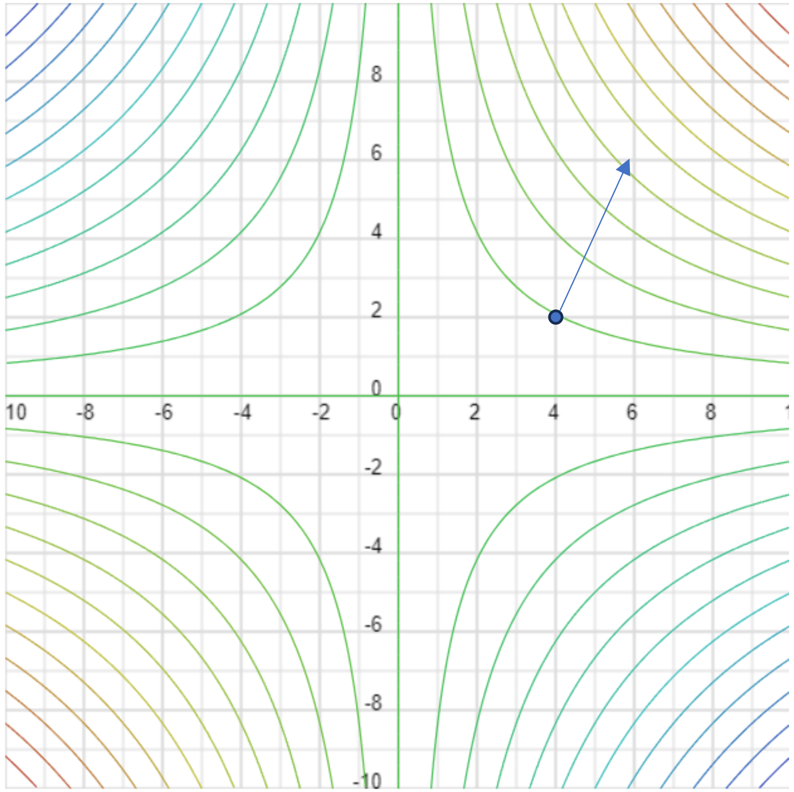
Descent

= Finding a way towards the minimum of the function



Gradient

- Gradient provides the direction of steepest ascent



$$f(x_1, x_2) = x_1 x_2$$

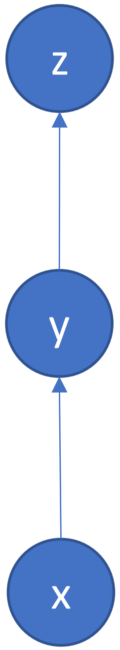
$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} = \begin{bmatrix} x_2 \\ x_1 \end{bmatrix}$$

Backpropagation

- We need to compute derivative of complex functions
- … but these are just chained simple functions

- Recursively apply chain rule

Chain rule



$$z = f(y)$$

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$$

$$y = f(x)$$

- Example

$$z = \sin(x^2)$$

Lets assume $y = x^2$, then we have

$$z = \sin(y)$$

$$\frac{\partial z}{\partial y} = \frac{\partial}{\partial y}\sin(y) = \cos(y) = \cos(x^2)$$

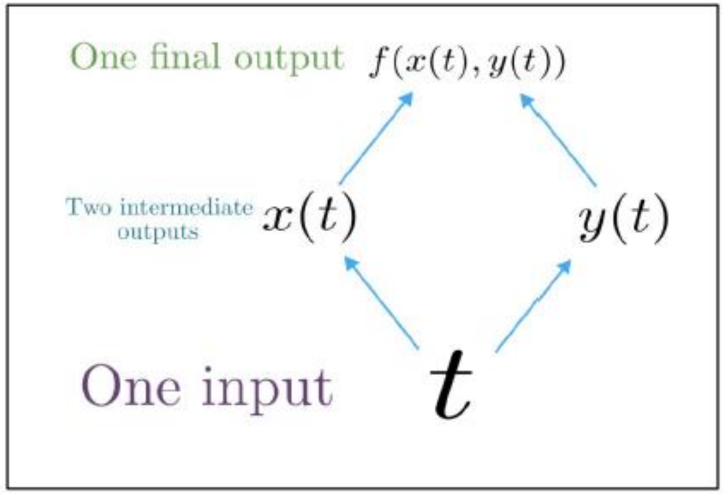
$$\frac{\partial y}{\partial x} = \frac{\partial}{\partial x}x^2 = 2x$$

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x} = \cos(x^2)\,2x$$

Chain rule (multi-variable)

- Given a multivariable function $f(x, y)$, and two single variable functions $x(t)$ and $y(t)$, here's what the multivariable chain rule says:

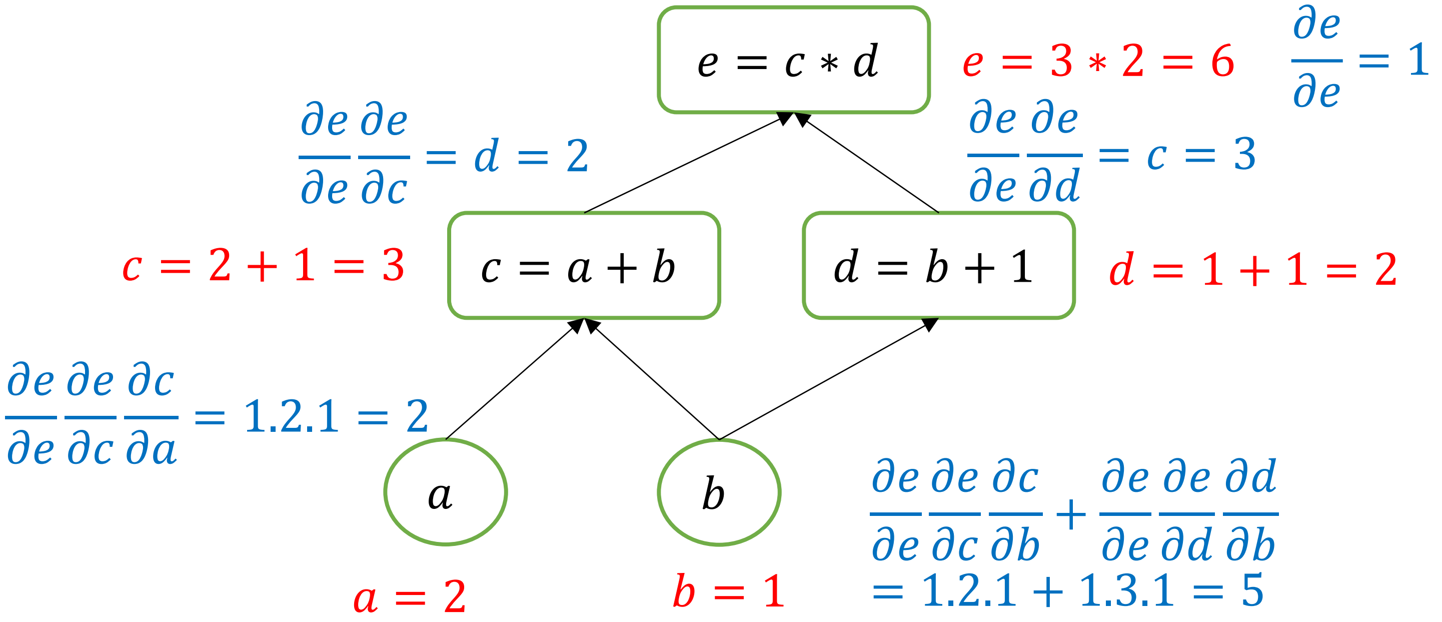
$$\underbrace{\frac{d}{dt} f(x(t), y(t))}_{\text{Derivative of composition function}} = \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt}$$



Computation graph

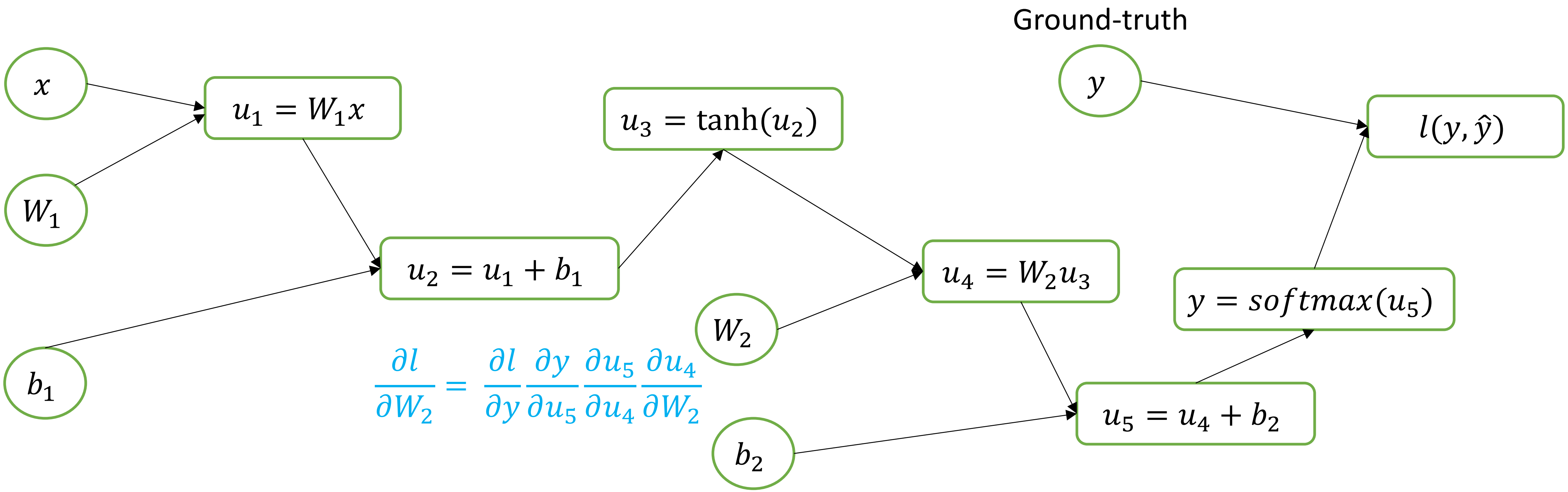
- The computations are organized as a graph

$$f(a,b) = (a+b) * (b+1)$$



Computation graph

$$\hat{y} = softmax(W_2 \tanh(w_1 x + b_1) + b_2)$$

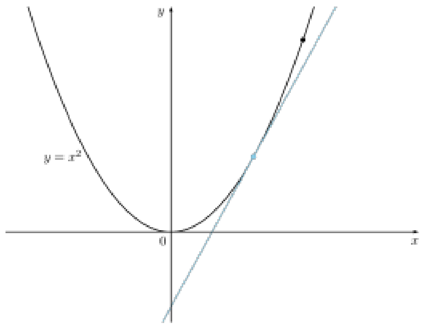


Gradient Descent

Gradient Descent

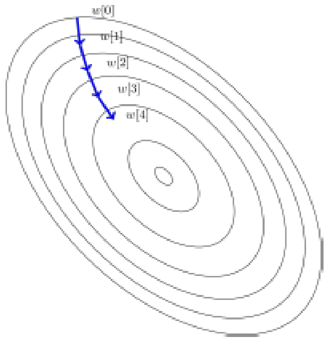
Gradient ✓

= The vector of all partial derivatives of a function



Descent

= Finding a way towards the minimum of the function



Gradient Descent

- Size of steps = Learning rate (η)
- Parameters of the model = θ
- Loss computed on the entire dataset = $J(\theta)$

$$\theta = \theta - \eta \underbrace{\nabla_\theta J(\theta)} \qquad \nabla_\theta J(\theta) = \frac{\partial}{\partial \theta} J(\theta)$$

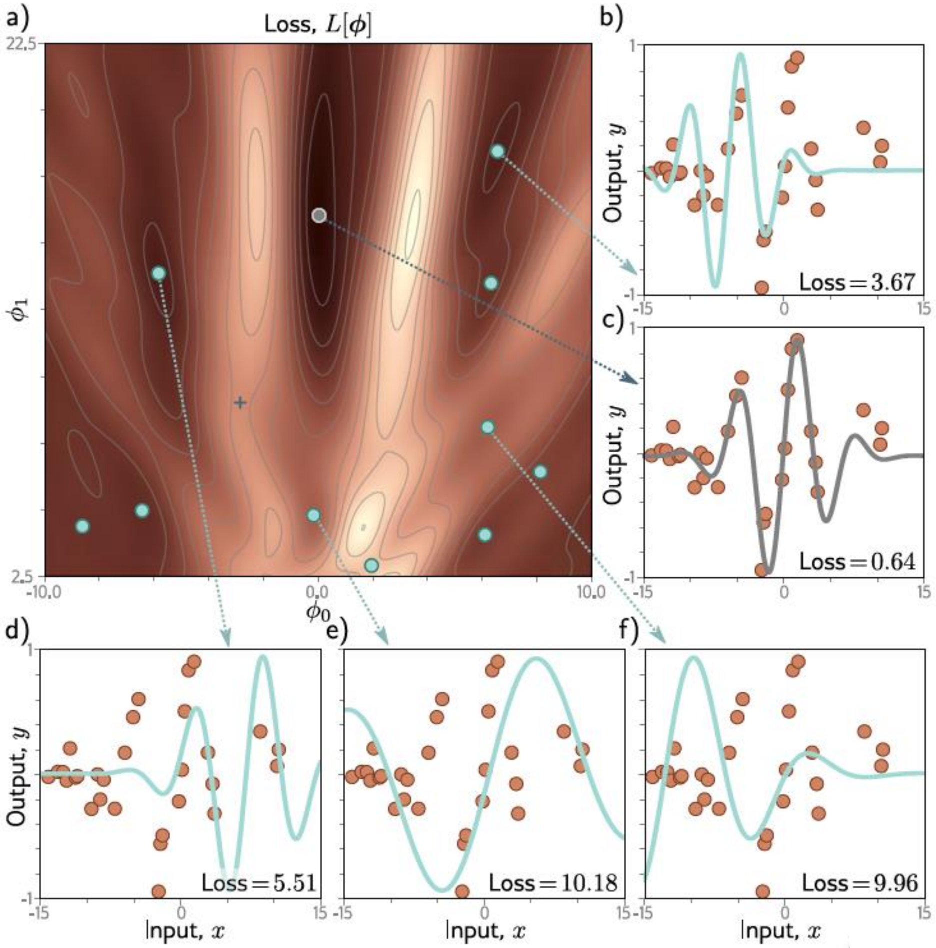
- Compute over multiple epochs

```
for i in range(nb_epochs):
  params_grad = evaluate_gradient(loss_function, data, params)
  params = params - learning_rate * params_grad
```

Gradient Descent

- GD is guaranteed to reach minima if the optimization function is convex
- It does not matter how you initialize the parameters
- Unfortunately, loss functions for non-linear models are non-convex

Local minima



Stochastic Gradient Descent

- Size of steps = Learning rate (η)
- Parameters of the model = θ
- At each step calculate objective function over a randomly chosen training example $\left(x^i, y^i\right)$
- Update for θ

$$\theta = \theta - \eta \nabla_\theta J(\theta; x^i, y^i)$$

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for example in data:
    params_grad = evaluate_gradient(loss_function, example, params)
    params = params - learning_rate * params_grad
```

Minibatch Gradient Descent

- Size of steps = Learning rate (η)
- Parameters of the model = θ
- At each step calculate objective function over a minibatch of n training examples
- Update for θ

$$\theta = \theta - \eta \nabla_\theta J(\theta; x^{i:i+n}, y^{i:i+n})$$

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```

Momentum

- Momentum: Helps accelerate SGD
- It does so by adding a fraction γ of the update vector of the past time step to the current vector

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$

$$\theta = \theta - v_t$$

Nesterov Momentum

- Nesterov Momentum
  - Momentum tends to overshoot the minimum -> Prevent that!
  - Use a lookahead: Take the gradient at the position that would be reached by the next step with the current velocity
- We know we will use momentum term γv_{t-1} to move the parameter θ
- $\theta - \gamma v_{t-1}$ gives us an approximation of where the parameter is going to be (lookahead)
- We calculate gradient with respect to our approximate future parameter instead of the current one

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$

$$\theta = \theta - v_t$$

Adagrad

- Introduces a cache variable that modifies the effective learning rate per parameter θ_i

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} g_{t,i}$$

$\nabla_\theta J(\theta_t, i)$

Sum of squares of past gradients with respect to θ

Normalize learning rate based on history

Adagrad

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} g_{t,i}$$

- Decreases learning rate for parameters that have large / frequent updates
- Increases learning rate for parameters that have small / infrequent updates
- Advantages:
  - NO (less) tuning of learning rates, (less sensitive to hyperparameter)
  - Faster convergence if scaling of the weights is unequal

Adam

- Adagrad with momentum

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

Consider both first and second moments

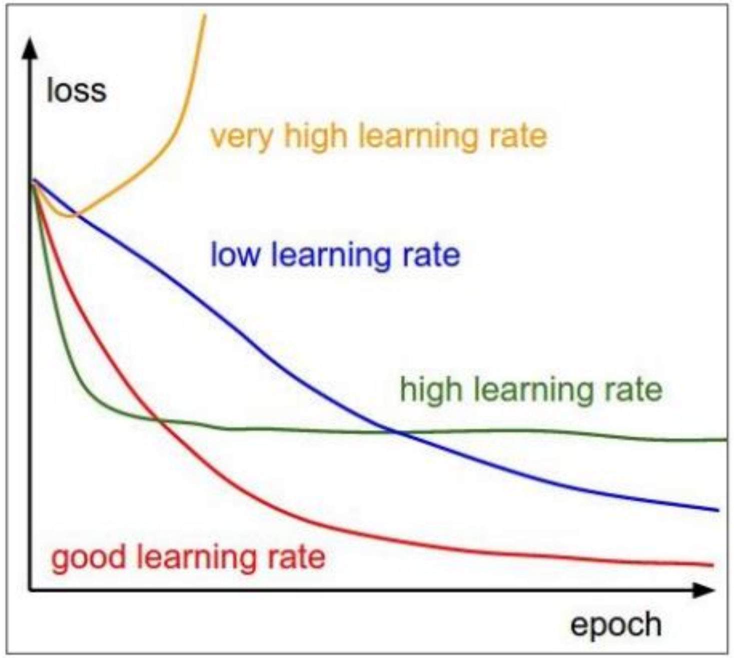
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

- m_t and v_t are initialized to 0 and are hence biased towards 0 in the initial steps
- Correct for such biases -

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \qquad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

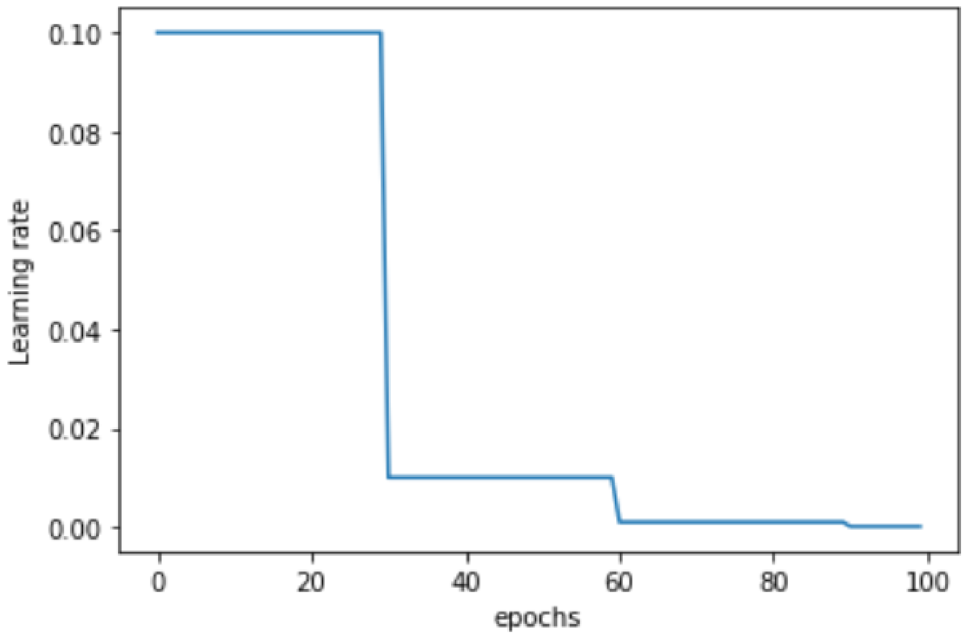
Learning rate

- SGD, SGD+momentum, Adagrad, Adam use learning rate as hyperparameter
- How do we choose a good learning rate?



Learning rate decays over time

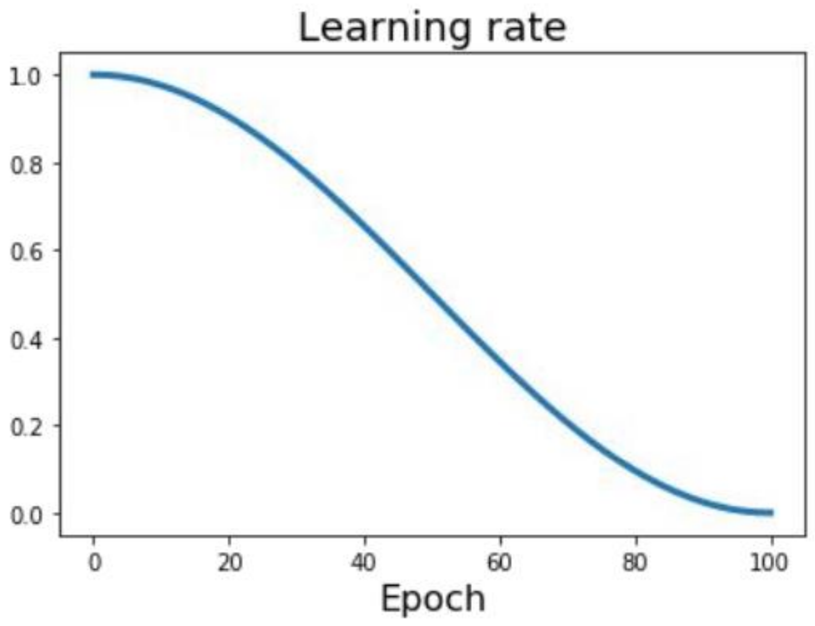
- Step: Reduce learning rate at a few fixed points
- E.g., multiply learning rate by 0.1 after 30, 60, 90 epochs



Learning Rate Decay

- Cosine

$$\alpha_t = \frac{1}{2}\alpha_0(1 + \cos(t\pi/T))$$

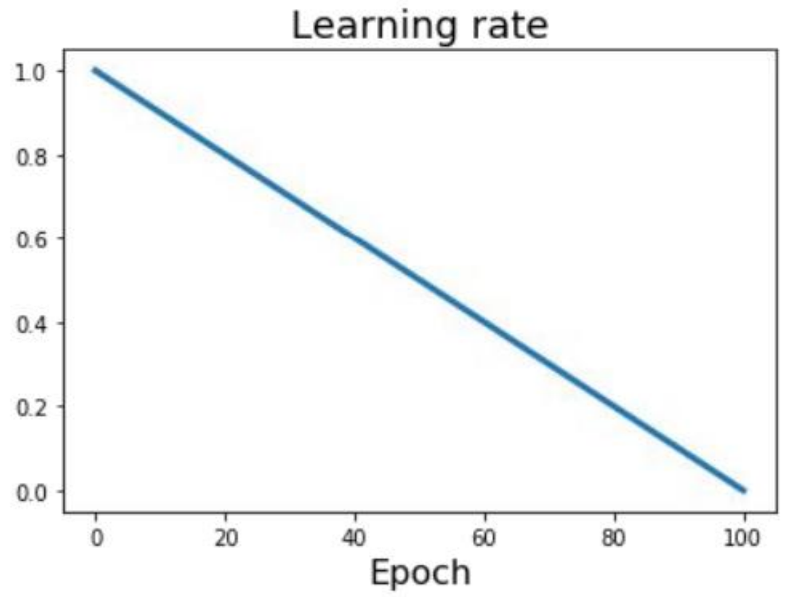


- α_0: Initial learning rate
- α_t: Learning rate at epoch t
- T: Total number of epochs

Learning Rate Decay

- Linear

$$\alpha_t = \alpha_0(1 - \mathrm{t/T})$$

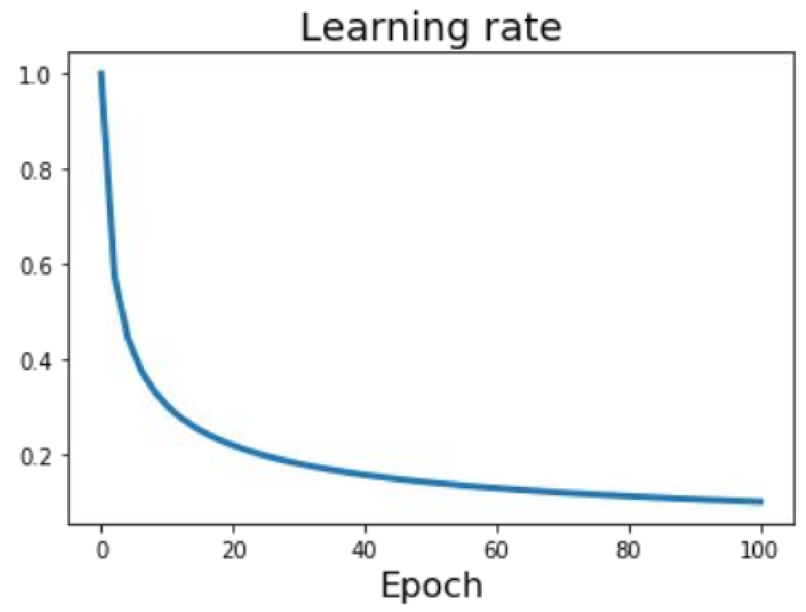


  - α_0: Initial learning rate
  - α_t: Learning rate at epoch t
  - T: Total number of epochs

Learning Rate Decay

- Inverted sqrt

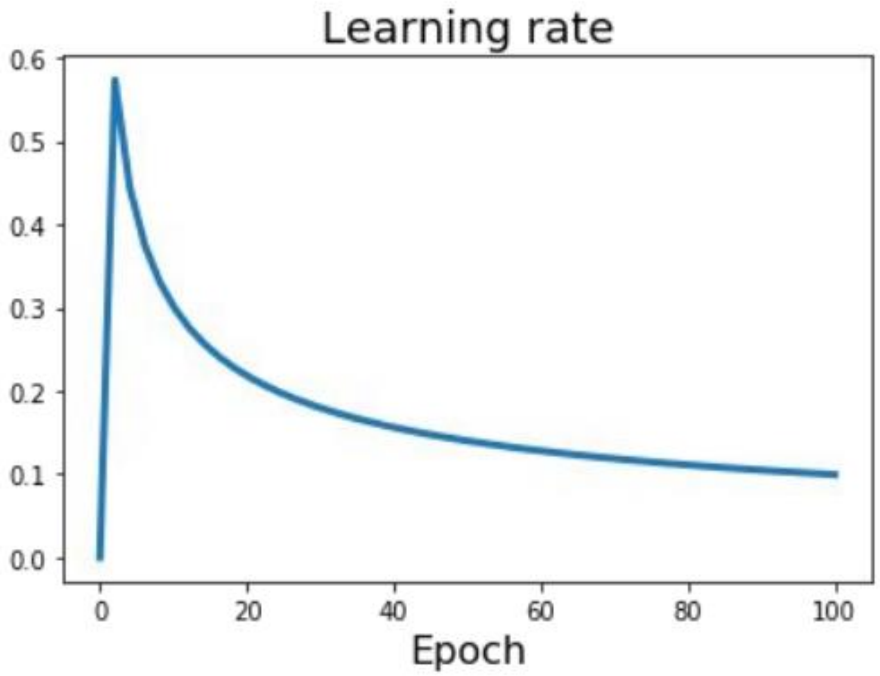
$$\alpha_t = \alpha_0/\sqrt{t}$$



- α_0: Initial learning rate
- α_t: Learning rate at epoch t
- T: Total number of epochs

Learning Rate Decay: Linear Warmup

- High initial learning rates can make loss explode
- Linearly increasing the learning rate from 0 over first ~ 5000 iterations can prevent this



Parameter Initialization

- How to initialize the parameters before we start training?

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

Parameter Initialization

- Initializing to 0 or to a constant
  - Hidden units will have identical influence on loss
  - Prevent different neurons from learning different things

- Initializing weights to very high values -> exploding gradient

- Initializing weights to very low values -> vanishing gradient

Appropriate Initialization

- Rules of thumb
  - The mean of the activations should be zero.
  - The variance of the activations should stay the same across every layer.

- Ensuring zero-mean and maintaining the value of the variance of the input of every layer guarantees no exploding/vanishing signal

Standard Normal

- Pick numbers from a standard normal distribution

```
W = 0.01*np.random.randn()
```

- More inputs lead to higher variance
- Can be fixed through normalization

$$w \sim N(0,1/n)$$

- n is the number of inputs to the neuron

Xavier Initialization

- Introduced by Xavier Glorot and Yoshua Bengio
- Recommend to normalize the variance to

$$Var(w) = \frac{2}{n_{in} + n_{out}}$$

Number of outputs

$$w \sim N(0, Var(w))$$

Number of inputs

Works best with sigmoid or tanh activation functions

Kaiming Initialization

- Introduced by Kaiming He et. Al

$$Var(w) = \frac{gain}{n_{in}}$$

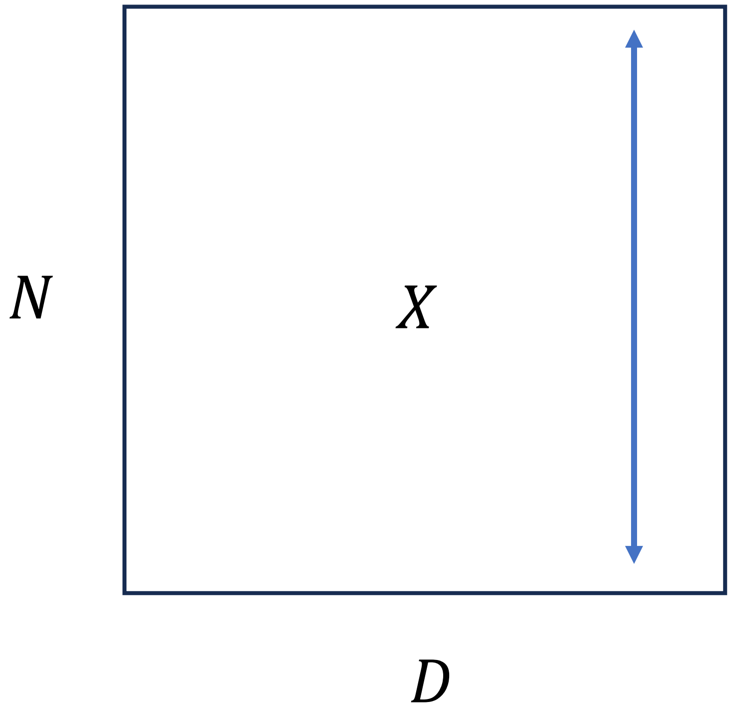
- $gain$ depends on the activation function (e.g., for ReLU, $gain$=2)

$$w \sim N(0, Var(w))$$

- Works best with ReLU

Batch Normalization

- Explicitly make the outputs Gaussian



$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N} \left(x_{i,j} - \mu_j\right)^2$$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Batch Normalization

- Learnable scale and shift parameters: γ, β

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N} \left(x_{i,j} - \mu_j\right)^2$$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}} \quad \Rightarrow \quad y_{i,j} = \gamma_j \, \hat{x}_{i,j} + \beta_j$$

Batch Normalization

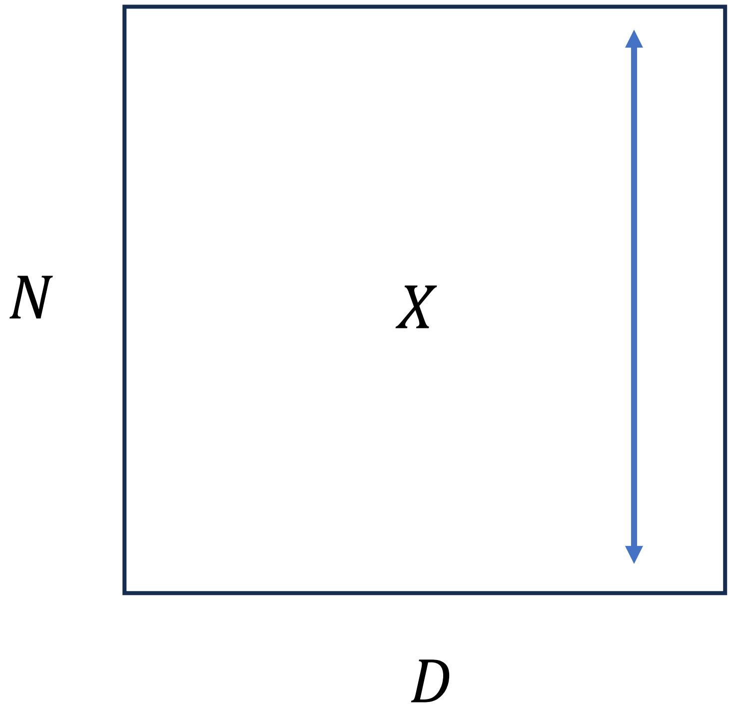
- During test time, μ_j, σ_j are not computed, rather an estimate from training is used

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

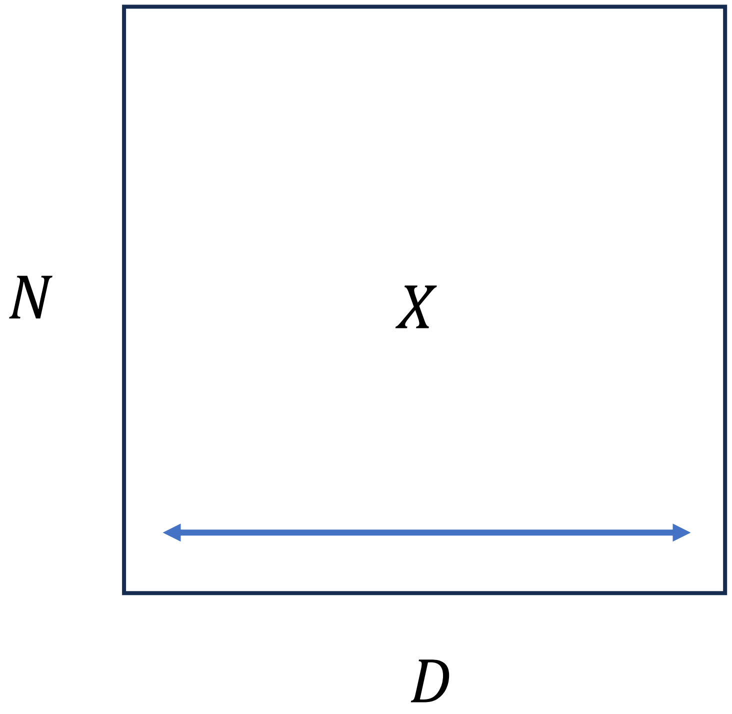
$$y_{i,j} = \gamma_j \, \hat{x}_{i,j} + \beta_j$$

Layer Normalization

- Same as batch norm only that the estimates are computed along dimensions



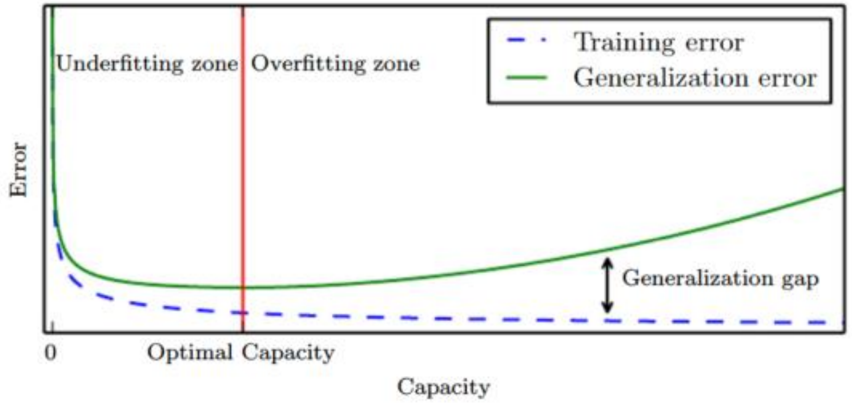
Batch Norm



Layer Norm

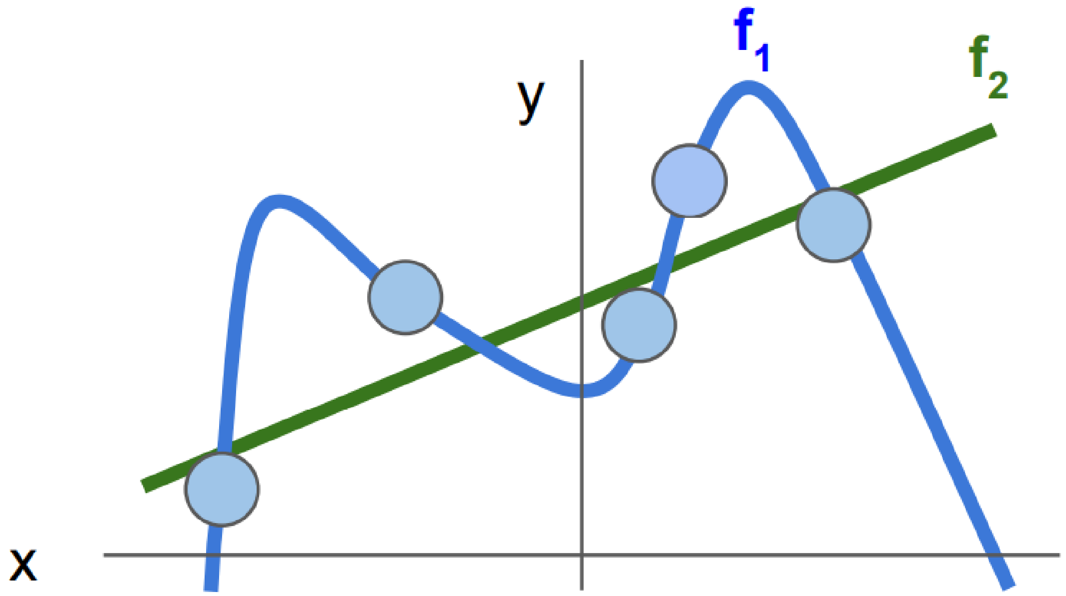
Regularization

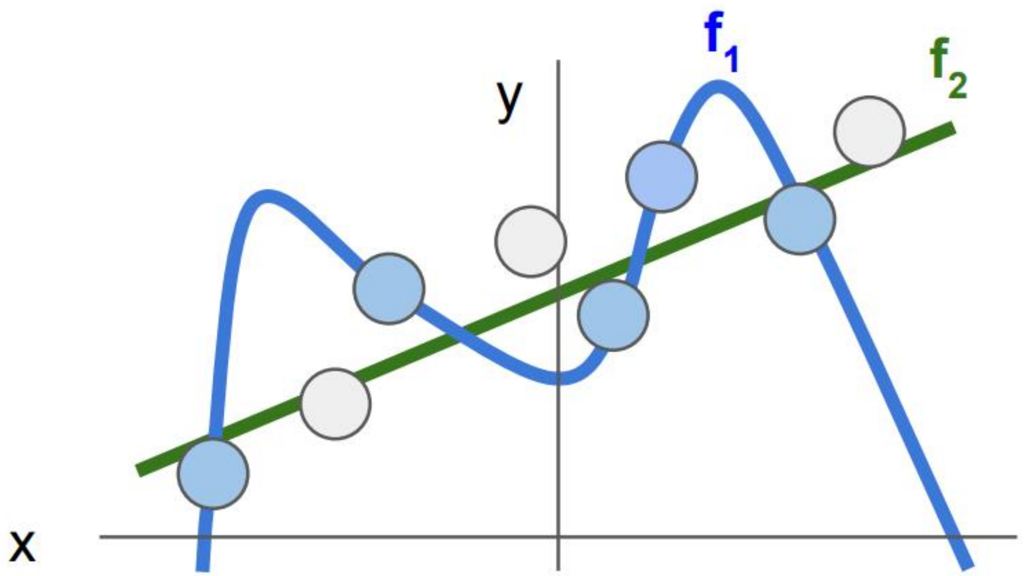
- Generalization/Test error
  - Performance on previously unseen inputs



- Regularization is:
  - Any modification to a learning algorithm to reduce its generalization error but not its training error
  - Reduce generalization error at the expense of test error

Regularization





Regularization

$$L(W) = \frac{1}{n}\sum_{i=1}^{n} l\left(f_W\left(x^i\right), y^i\right) + \lambda R(W)$$

- Data loss: Model predictions should match training data
- Regularization: Prevent the model from doing too well on the training data

Regularization

$$L(W) = \frac{1}{n}\sum_{i=1}^{n} l\left(f_W\left(x^i\right), y^i\right) + \lambda R(W)$$

- L2 regularization:

$$R(W) = \frac{1}{2}\sum |W|^2$$

- L1 regularization:

$$R(W) = \sum |W|$$

Regularization

- Expressing preference over weights

$$x = [1, 1, 2, 1]$$

$$W_1 = [0, 0, 1, 0]$$

$$W_2 = [0.5, 0.5, 0.25, 0.5]$$

$$W_1^T x = W_2^T x = 2$$

Which W will L_2 prefer?

Which W will L_1 prefer?

Regularization

- L_1 prefers weights which are sparse
- L_2 prefers weights which are more spread out
- Decide on which regularization to use depending on the task.
- L_2 is used more often

Regularization

- Elastic net ($L_1 + L_2$)

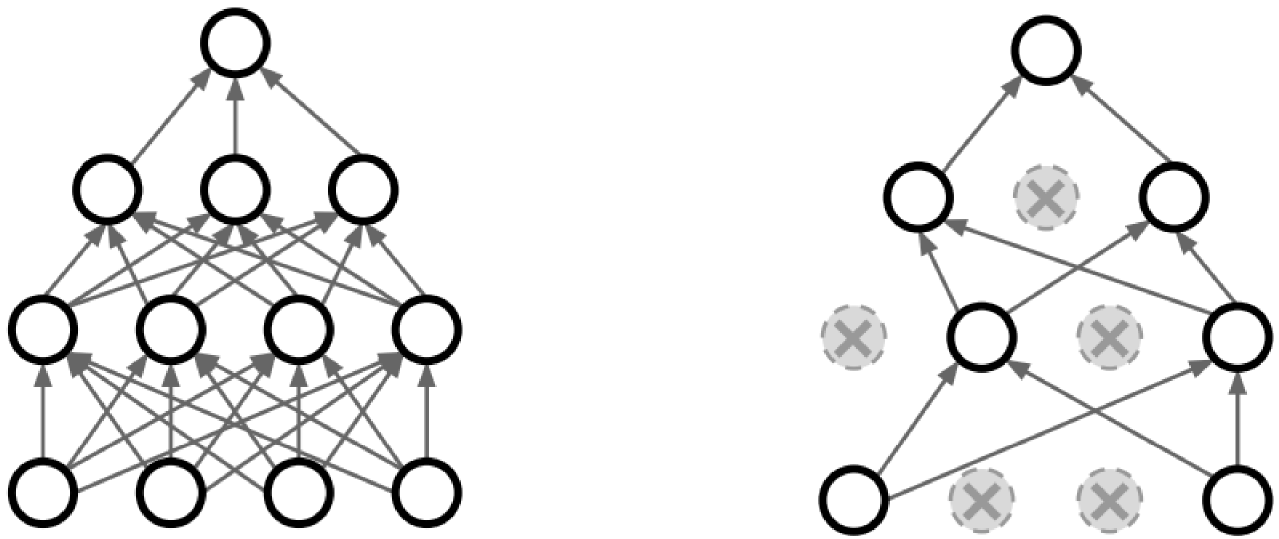
$$R(W) = |W| + \beta|W|^2$$

- Weight decay: Adds a regularization term to the gradient of loss

$$\frac{\partial}{\partial W}L + \lambda W$$

Regularization

- Dropout: In each forward pass, randomly set some neurons to Zero



- Force the model to not rely on only a certain set of features

Regularization

- Dropout makes our output random

$$y = f_W(x, z)$$ Random mask

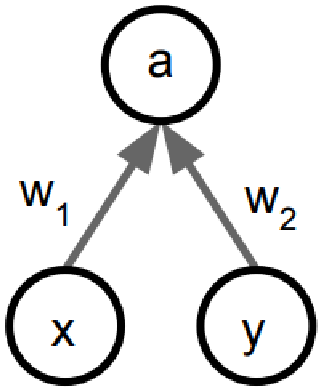
- Want to "average out" the randomness at test time

$$y = f(x) = E_z[f_W(x, z)] = \int p(z) f_W(x, z) dz$$

Difficult to compute…

Regularization

- We would like to approximate the integral



At test: $E[a] = W_1x + W_2y$

During training:

$$E[a] = \frac{1}{4}(W_1x + W_2y) + \frac{1}{4}(W_1x + 0y) + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + W_2y)$$

$$= \frac{1}{2}(W_1x + W_2y)$$

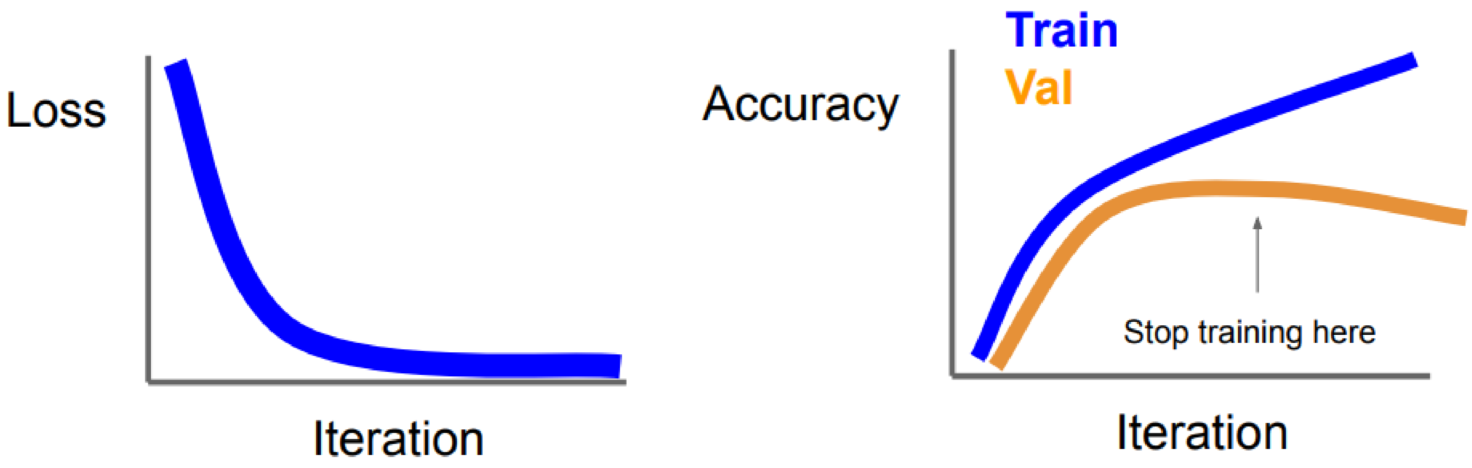
At test time multiply with the dropout probability

Regularization

- Inverted dropout
- Normalize by p during training
- Test time is unchanged

Early Stopping

- Stop training the model when accuracy on the validation set decreases Or train for a long time, but always keep track of the model snapshot that worked best on val



Summary

- Training neural networks
  - Optimization
  - Weight Initialization
  - Regularization

References

- Regularization slides adopted from CS231n course at Stanford
- https://www.ruder.io/optimizing-gradient-descent/ (Optimization)
- https://colah.github.io/posts/2015-08-Backprop/