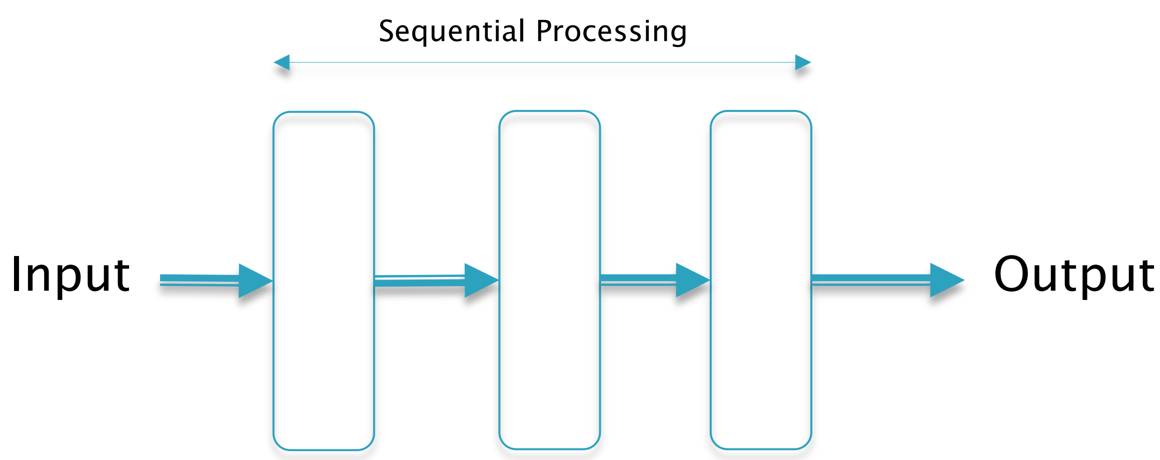
Training Process Improvements: Part 1

Lecture 7

Subir Varma

Network Topologies for Deep Networks

Sequential Processing

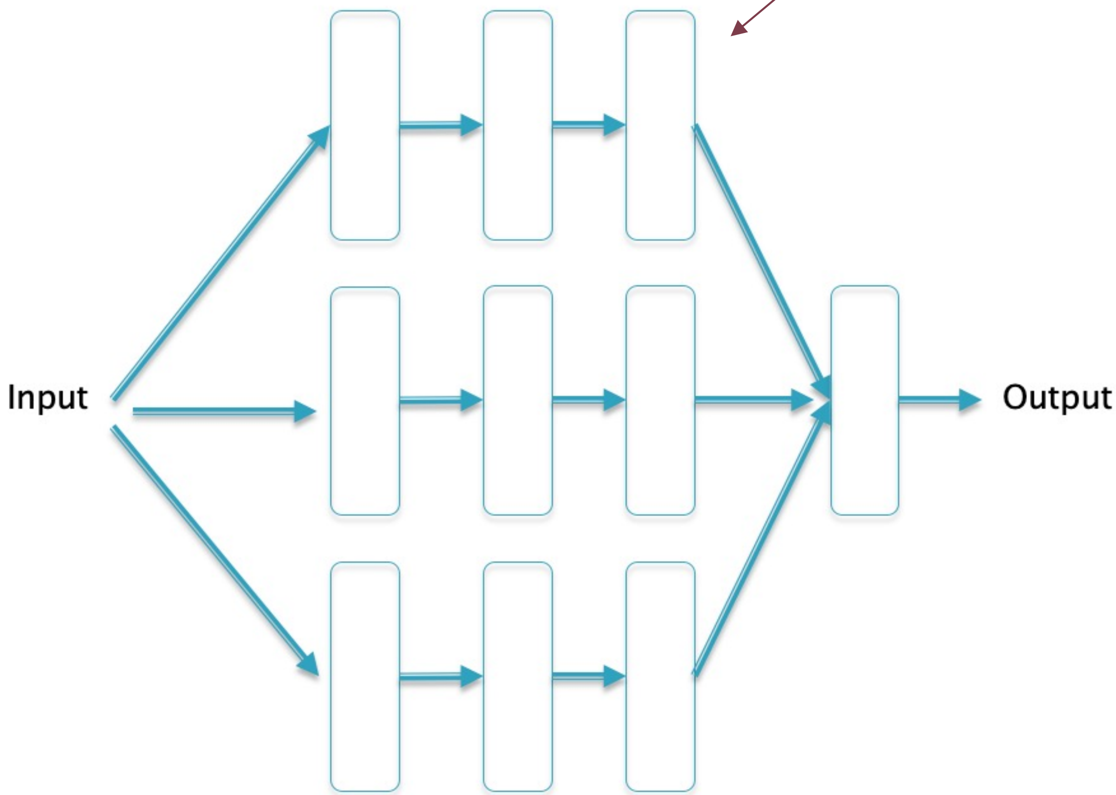


Using Sequential API

```
import keras
keras.__version__
from keras import models
from keras import layers

from keras.datasets import cifar10

(train_images, train_labels), (test_images, test_labels) = cifar10.load_data()

train_images = train_images.reshape((50000, 32 * 32 * 3))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 32 * 32 * 3))
test_images = test_images.astype('float32') / 255

from tensorflow.keras.utils import to_categorical

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)

network = models.Sequential()
network.add(layers.Dense(20, activation='relu', input_shape=(32 * 32 * 3,)))
network.add(layers.Dense(15, activation='relu'))
network.add(layers.Dense(10, activation='softmax'))

network.compile(optimizer='sgd',
                loss='categorical_crossentropy',
                metrics=['accuracy'])

history = network.fit(train_images, train_labels, epochs=100, batch_size=128, validation_split=0.2)
```

Parallel Processing: Model Ensembles

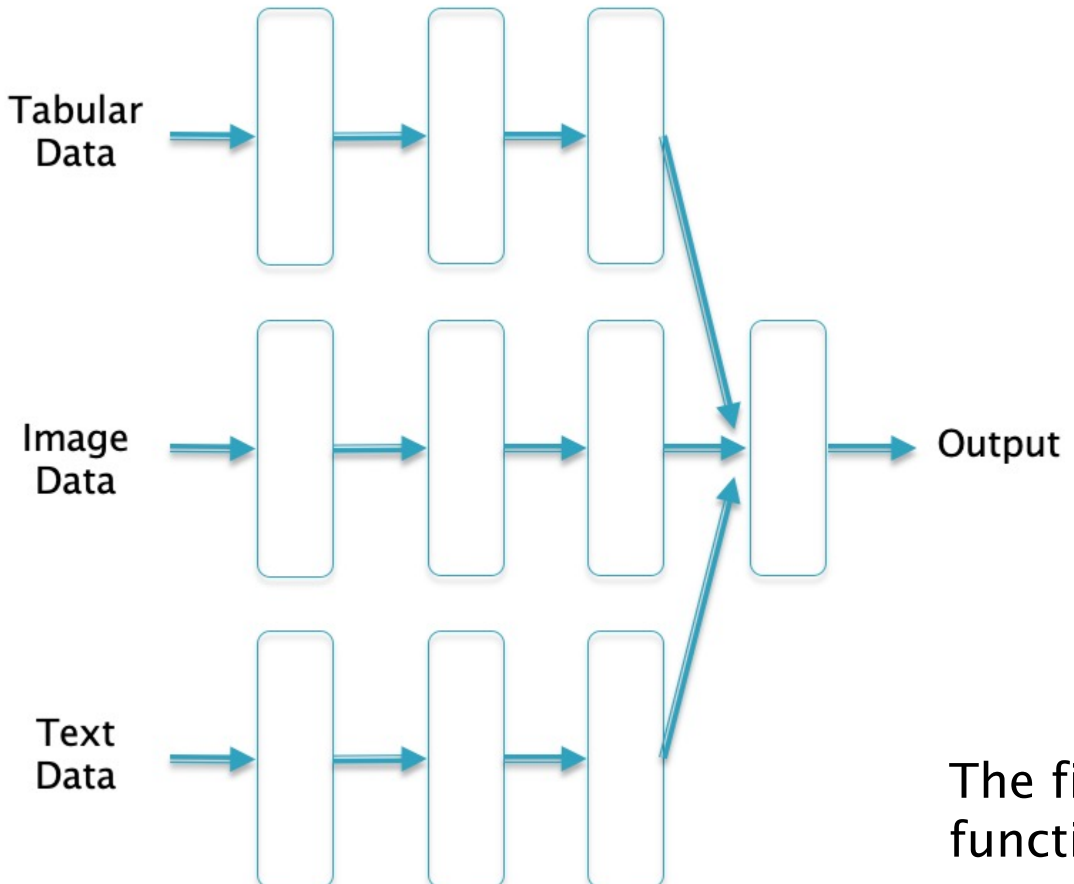
Multiple Predictions



Example: Majority Vote models

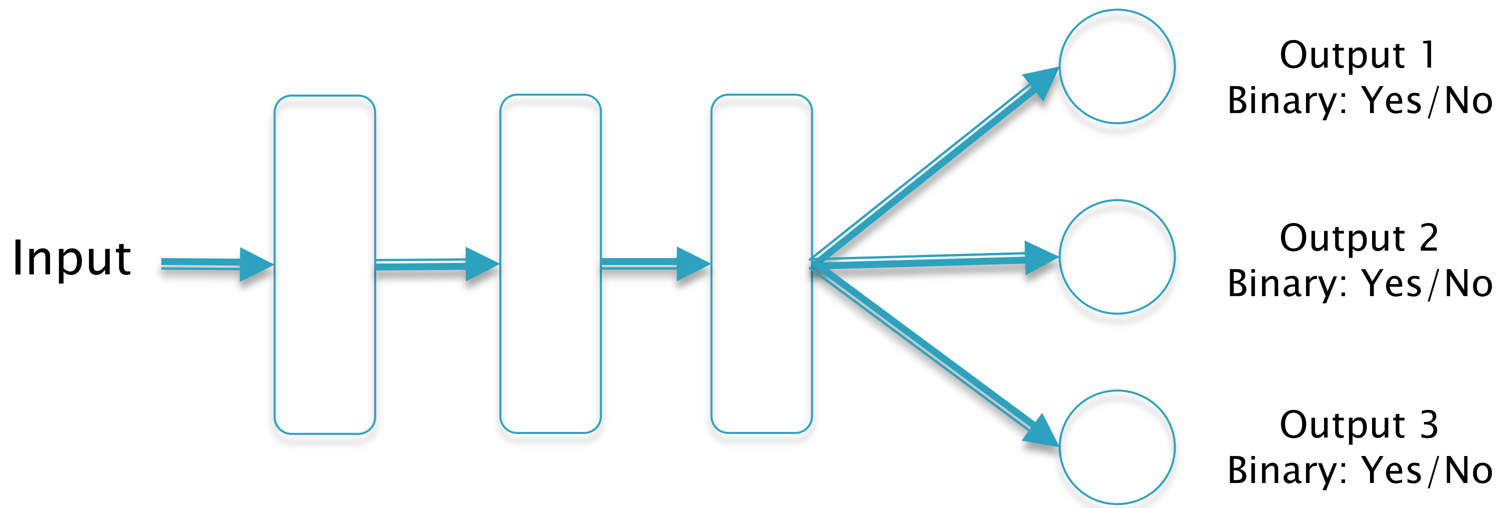
Increases prediction accuracy

Multi-Input Models



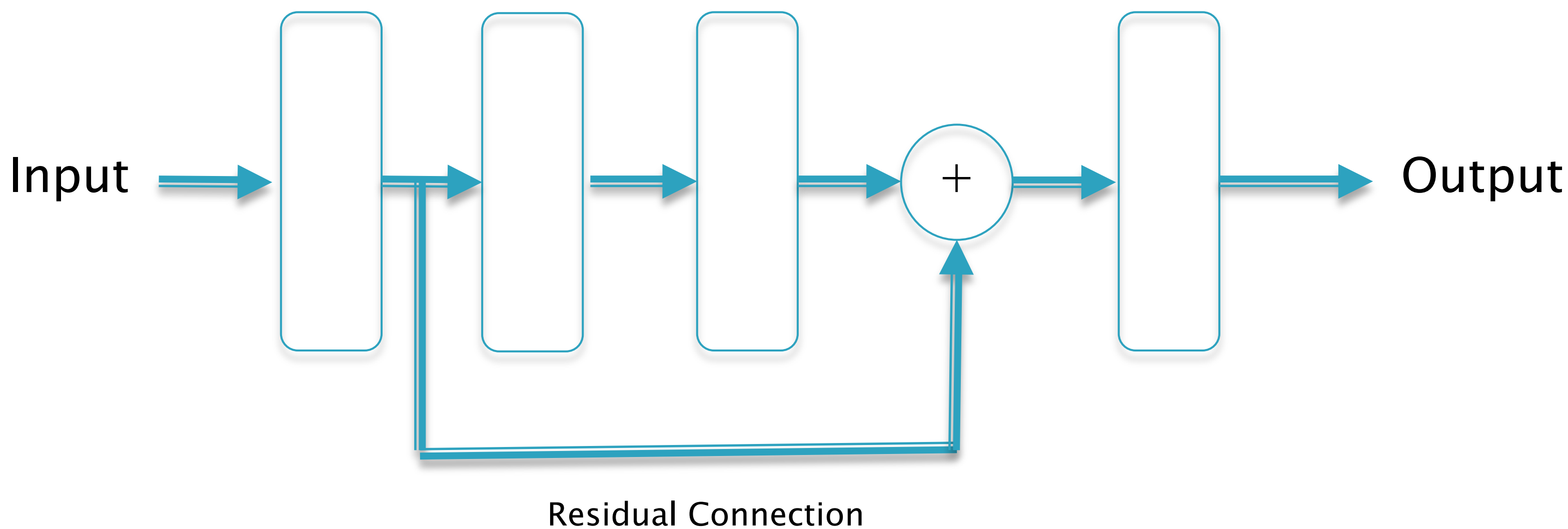
The final decision is a function of more than one type of input data

Multi-Label Classification



For classifying more than one object per input

Residual Connections



Enables the training of models with hundreds of hidden layers

Keras Functional API

All these different topologies can be easily coded using the Keras Functional API

```
import keras
keras.__version__
from keras import Sequential, Model
from keras import layers
from keras import Input

from keras.datasets import cifar10

(train_images, train_labels), (test_images, test_labels) = cifar10.load_data()

train_images = train_images.reshape((50000, 32 * 32 * 3))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 32 * 32 * 3))
test_images = test_images.astype('float32') / 255

from tensorflow.keras.utils import to_categorical

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)

input_tensor = Input(shape=(32 * 32 * 3,))
x = layers.Dense(20, activation='relu')(input_tensor)
y = layers.Dense(15, activation='relu')(x)
output_tensor = layers.Dense(10, activation='softmax')(y)

model = Model(input_tensor, output_tensor)

model.compile(optimizer='sgd',
                loss='categorical_crossentropy',
                metrics=['accuracy'])

history = model.fit(train_images, train_labels, epochs=10, batch_size=128, validation_split=0.2)
```

Keras Callbacks

- Model Checkpointing: Saving the current state of the model at different points during training
- Early Stopping: Interrupting Training when the Validation Loss is no longer improving (and saving the best model)
- Dynamically adjusting hyper parameter values: Example Learning Rate
- Logging Training and Validation Metrics

Training Process Issues

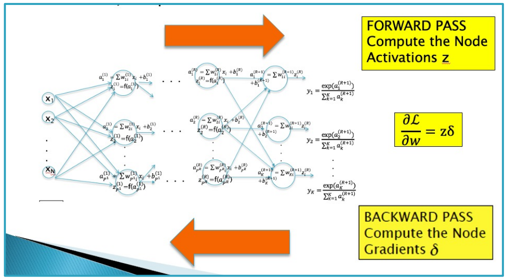
What we know so far?

- We defined a Dense Feed Forward Network Model that is a generalization of Linear Logistic Regression Models

- We defined a Training Algorithm to iteratively estimate model parameters using Stochastic Gradient Descent

$$w_{ij} \leftarrow w_{ij} - \eta \frac{\partial \mathcal{L}}{\partial w_{ij}}$$

- We discussed the Backprop algorithm, which is a fast and efficient way to compute the gradients with respect to the model parameters



What's Left?

Training using Backpropagation was known by the mid-1980s
Yet it took 20+ years for the field to make progress

Why?

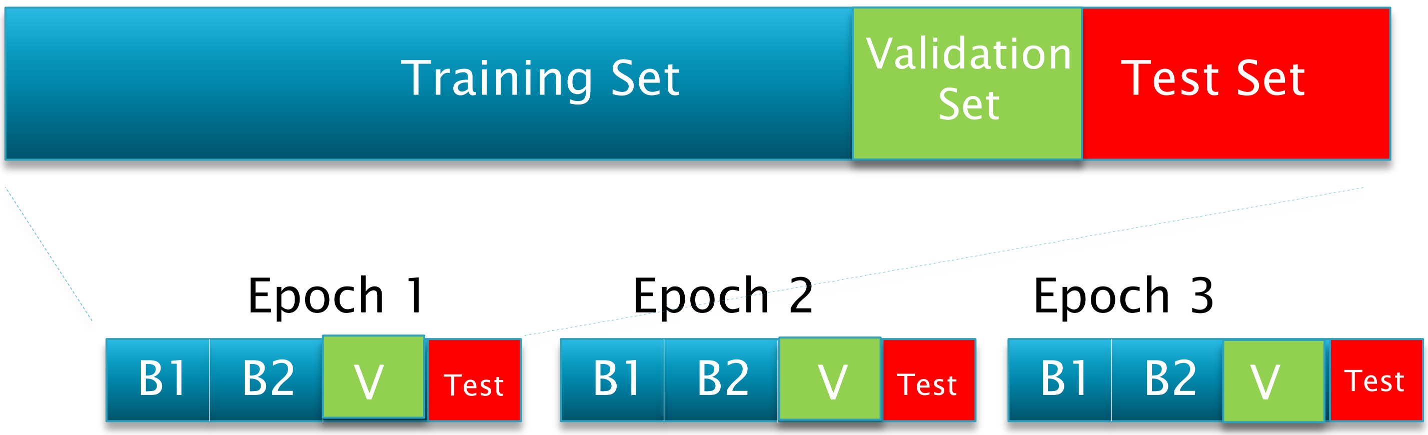
Backpropagation Did Not Work Very Well for Large Models

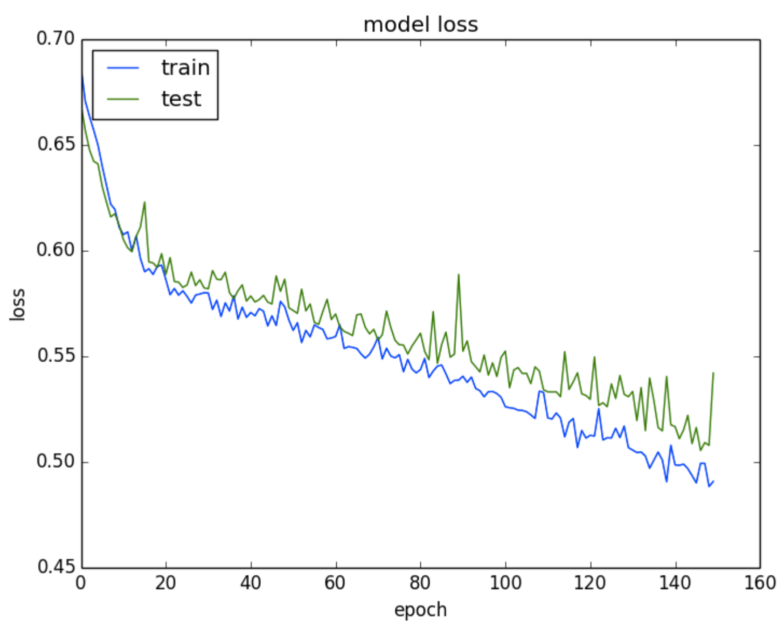
Culprit: The Vanishing Gradient Problem

Training Process Improvements

1. Not all Activation Functions work well
2. Stochastic Gradient Descent can be improved upon
3. How can a model's generalization capabilities be improved?
4. How to choose good values for hyper-parameters?
5. How to initialize the weight parameters properly?
6. The stopping problem: When to stop the training process?

Training, Validation and Test Sets





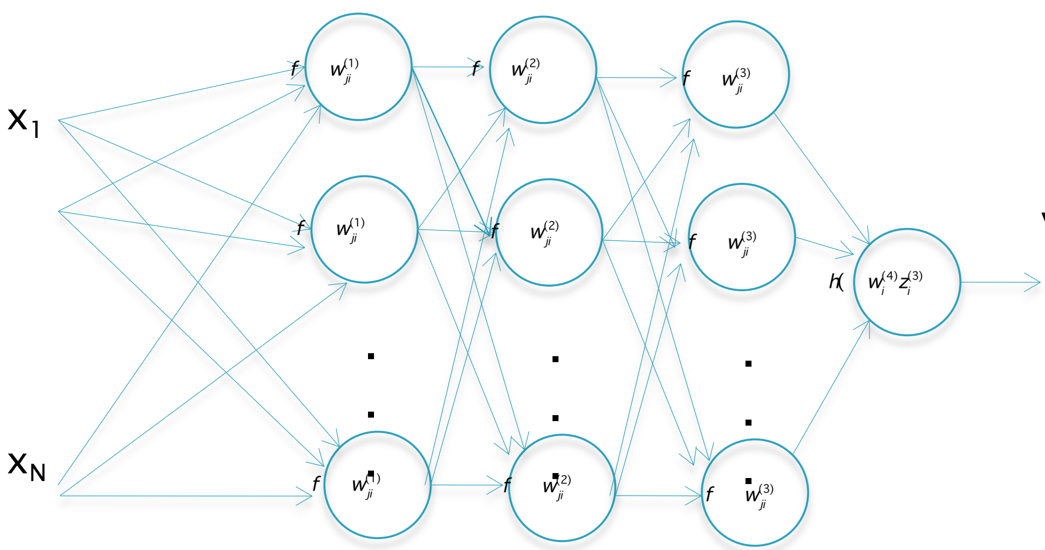
- Test Dataset:
  - Not used for Training
  - Not used for choosing model parameters
- Validation Dataset:
  - Not used for Training
  - Used for choosing model parameters

Why is a Validation Data Set Needed?

- Validation Data Set allows us to experiment with Hyper-Parameter settings
- Why don't we use the Test Data Set to experiment with Hyper-Parameter values?
  - By doing so, we may end up finding hyper-parameters which fit particular peculiarities of the Test Data, but where the performance of the network won't generalize to other Test data sets

Vanishing Gradient Problem

The Vanishing Gradient Problem: Definition



Compute Error Signal

$$\delta = \frac{\partial L}{\partial a} = y - t$$

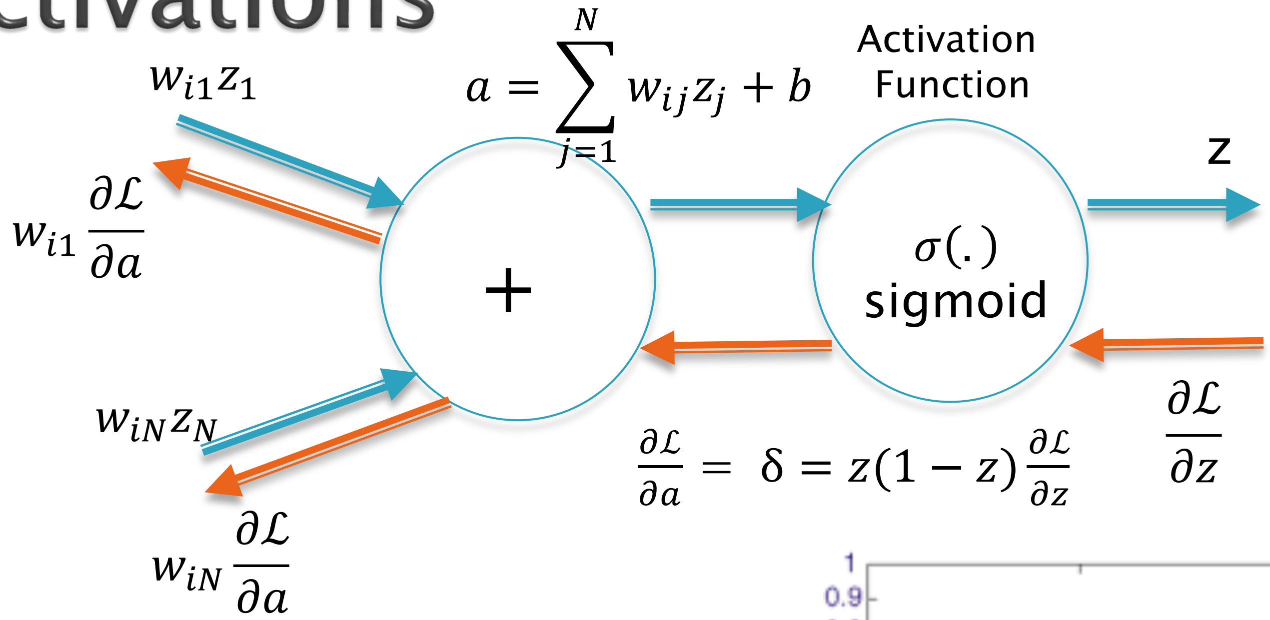
And Propagate It Backwards

Problem: The gradient δ would die to 0 after a few layers

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = z_j \delta_i$$

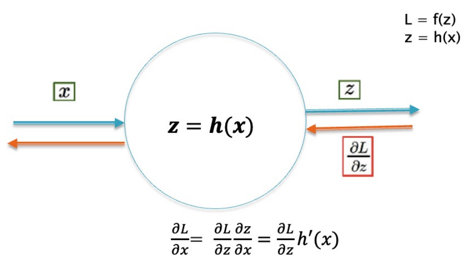
$$w_{ij} \leftarrow w_{ij} - \eta \frac{\partial \mathcal{L}}{\partial w_{ij}}$$

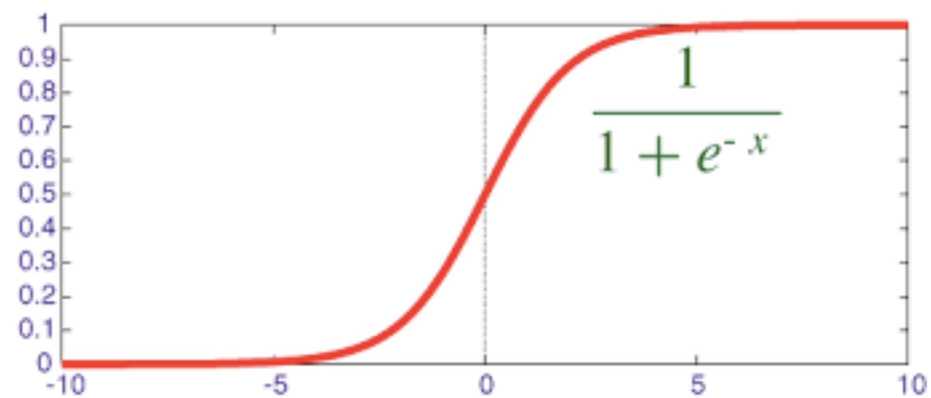
The Problem with Sigmoid Activations



$$\boxed{\frac{\partial \mathcal{L}}{\partial w_{ij}} = x_j \delta_i}$$

$$\boxed{w_{ij} \leftarrow w_{ij} - \eta \frac{\partial \mathcal{L}}{\partial w_{ij}}}$$





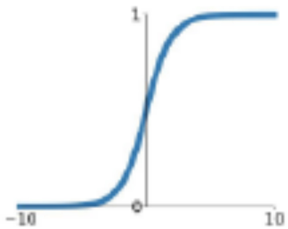
If a node is saturated (i.e. |a| >>0), then its gradient will go to zero, and all the weights incident on that node will stop adapting.

Activation Functions

Activation Functions

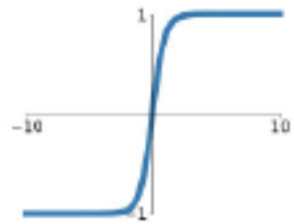
Sigmoid

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



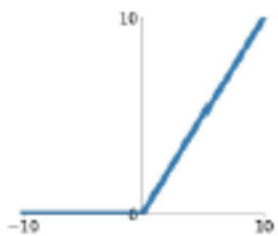
tanh

$\tanh(x)$



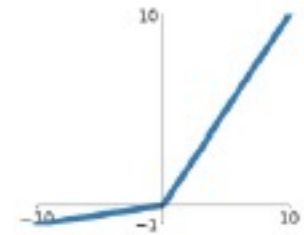
ReLU

$\max(0, x)$



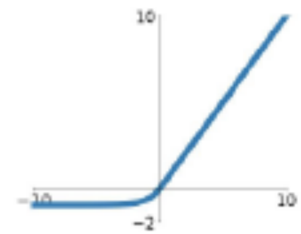
Leaky ReLU

$\max(0.1x, x)$



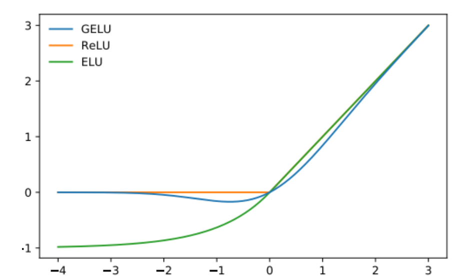
ELU

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

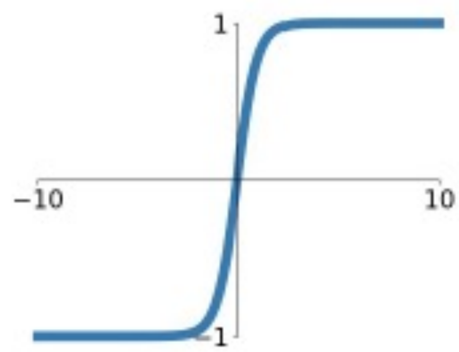


GeLU

$x\Phi(x)$



Activation Functions: tanh

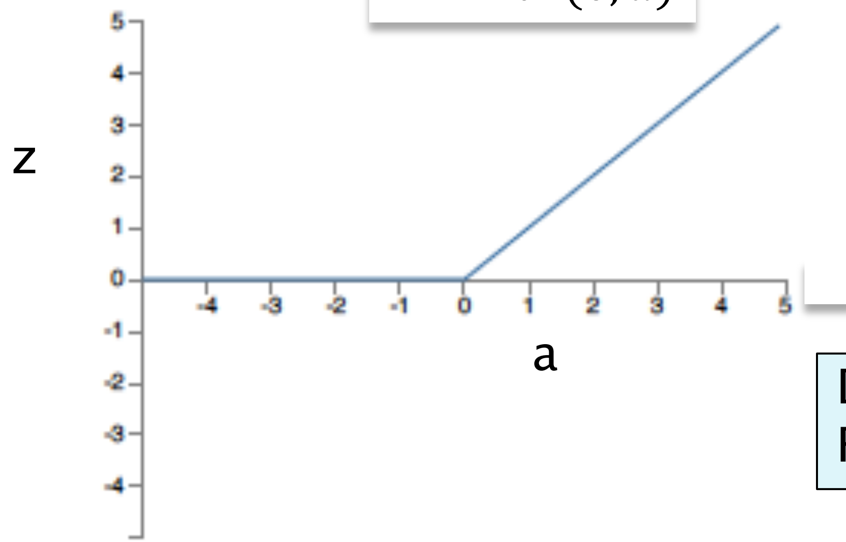


$$z = \tanh a = \frac{e^{a} - e^{-a}}{e^{a} + e^{-a}}$$

- Squashes inputs into the range [−1,+1]
- Does not have the zero centering issue
- However, still saturates for values away from zero

Activation Functions: ReLU

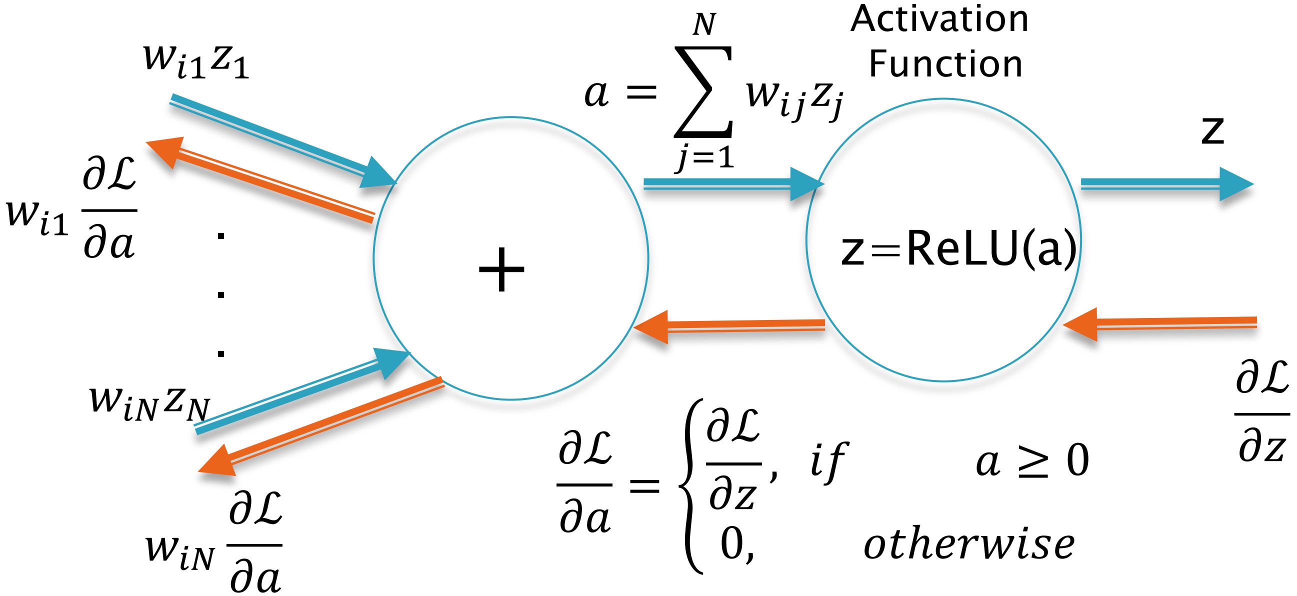
$z = \max(0, a)$



Default Choice
For Activation Function

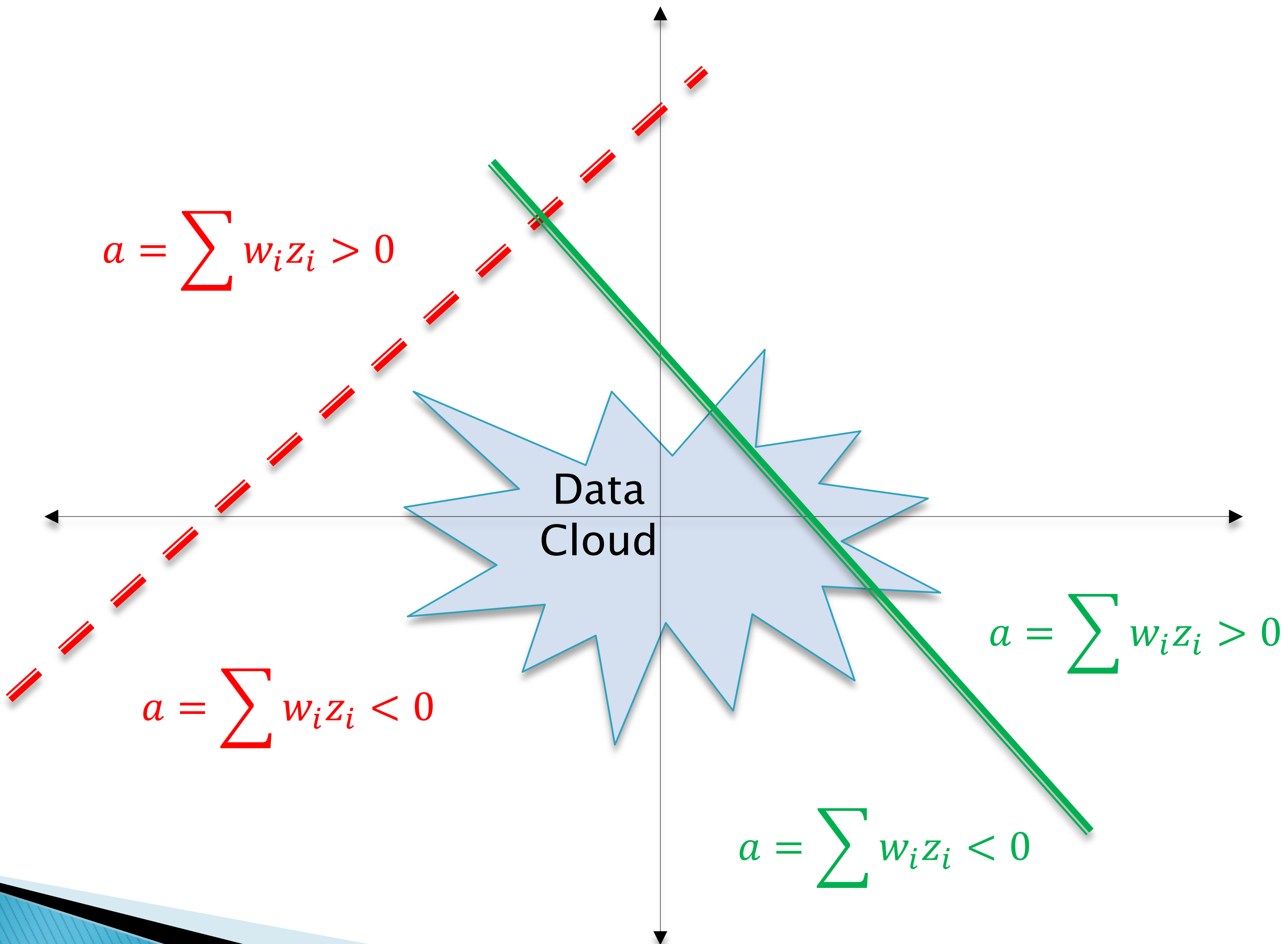
- Very computationally efficient
- Acts as a Gate: If a>0 then it passes gradient through, if a<0 then it kills the gradient.
- Leads to much faster convergence (by a factor of 6 in the AlexNet case), Possible Reasons
  - Does not saturate (for a>0) for half of the input range
- Issues:
  - Not Zero Centered

ReLU (cont)



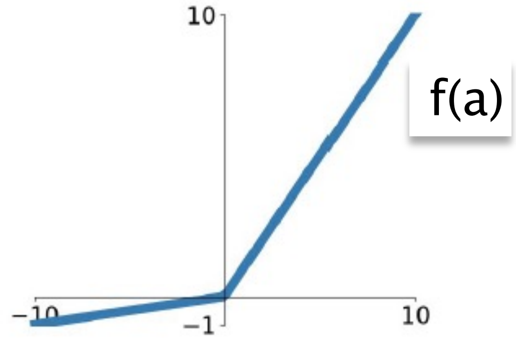
$w_{i1}z_1$
$w_{i1}\frac{\partial \mathcal{L}}{\partial a}$
$w_{iN}z_N$
$w_{iN}\frac{\partial \mathcal{L}}{\partial a}$
+
$a = \sum_{j=1}^{N} w_{ij}z_j$
Activation Function
z=ReLU(a)
z
$\frac{\partial \mathcal{L}}{\partial z}$
$\frac{\partial \mathcal{L}}{\partial a} = \begin{cases} \frac{\partial \mathcal{L}}{\partial z}, & if \quad a \geq 0 \\ 0, & otherwise \end{cases}$

The Dead ReLU Problem



Activation Functions: Leaky ReLU

Leaky ReLU
f(a) = max(ca,a)

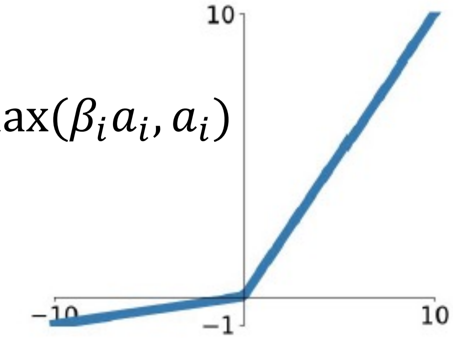


- Has all the advantages of ReLU + Does not saturate

$$\frac{\partial \mathcal{L}}{\partial a} = \begin{cases} \frac{\partial \mathcal{L}}{\partial z}, \; if & a \geq 0 \\ c\frac{\partial \mathcal{L}}{\partial z}, & otherwise \end{cases}$$

Activation Functions: Pre ReLU

$z_i = \max(\beta_i a_i, a_i)$



$f(x)$

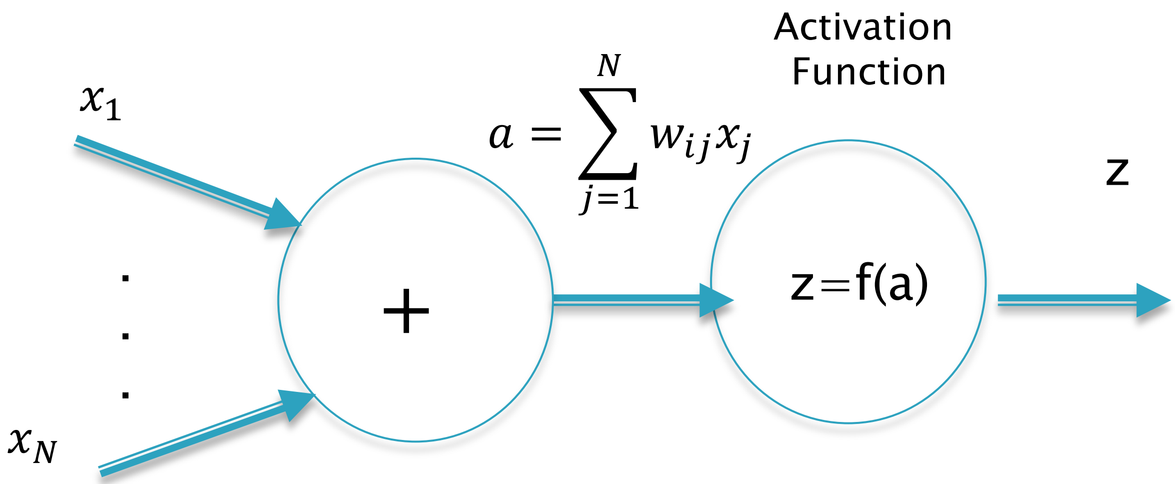
- The parameter beta is learnt during the training process using backprop

$$\frac{\partial \mathcal{L}}{\partial \beta_i} = \frac{\partial \mathcal{L}}{\partial z_i} \frac{\partial z_i}{\partial \beta_i}$$

$$\frac{\partial \mathcal{L}}{\partial \beta} = \begin{cases} 0 & if \qquad \beta \leq 1 \\ a \dfrac{\partial \mathcal{L}}{\partial z}, & otherwise \end{cases}$$

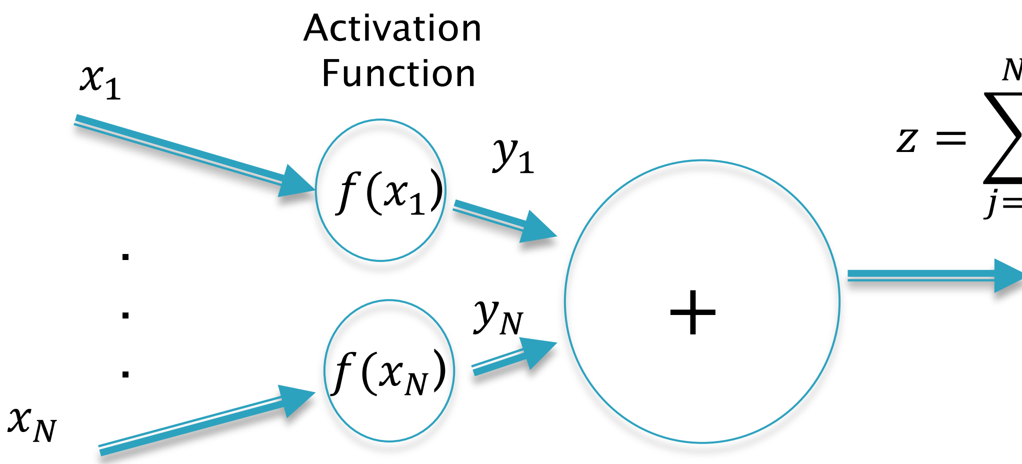
$$\beta \leftarrow \beta - \eta \frac{\partial \mathcal{L}}{\partial \beta}$$

Traditional Activation



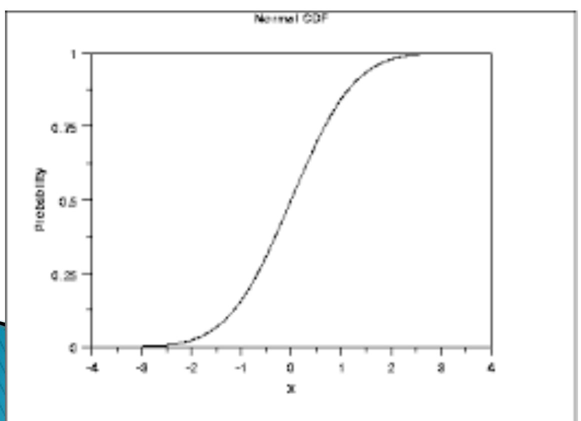
GeLU Activation

z is a probabilistic function



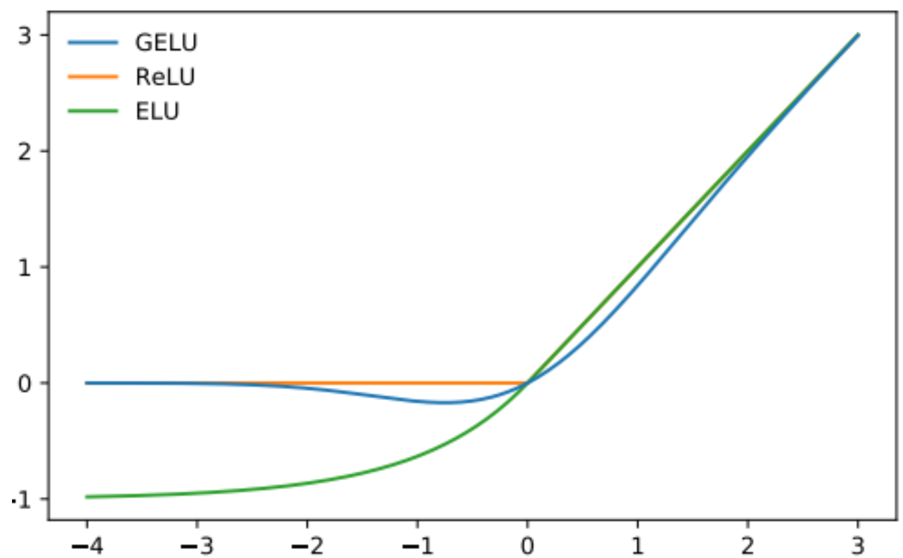
$$z = \sum_{j=1}^{N} w_{ij} y_j$$

$$y = f(x) = \begin{cases} x & \text{with probability } \Phi(x) \\ 0 & \text{with probability } 1 - \Phi(x) \end{cases}$$



Standard Normal Distribution for N(0,1)

GeLU Activation



Summary

- Use ReLU or GeLU
- Try out Leaky ReLU, PreLU, MaxOut
- Don’t use Sigmoid
- Try out tanh but don’t expect much improvement

Activation Functions in Keras

```
from keras import models
from keras import layers

network = models.Sequential()
network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
network.add(layers.Dense(10, activation='softmax'))
```

Information available at:

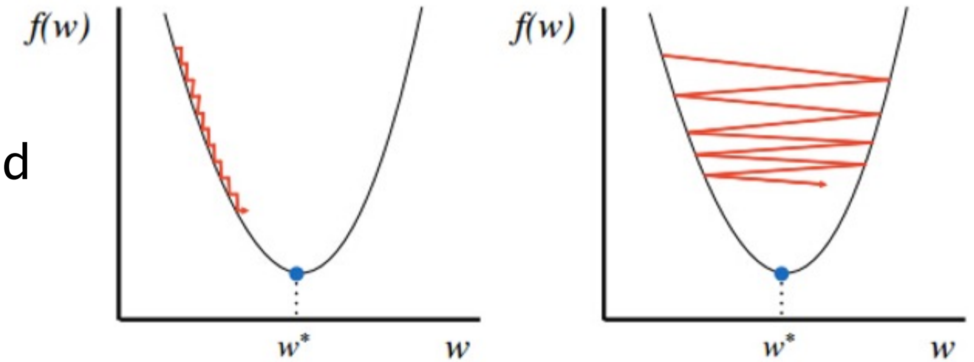
https://keras.io/activations/

Improved SGD

Issues with SGD

$$w_{ij} \leftarrow w_{ij} - \eta \frac{\partial \mathcal{L}}{\partial w_{ij}}$$

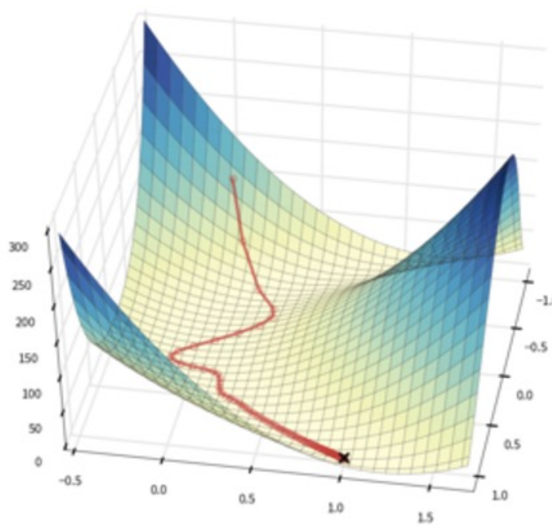
Choosing a good Learning Rate η

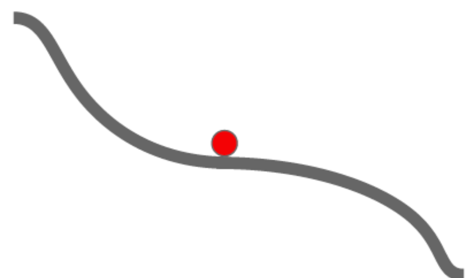


Too small: converge very slowly

Too big: overshoot and even diverge

Saddle Points



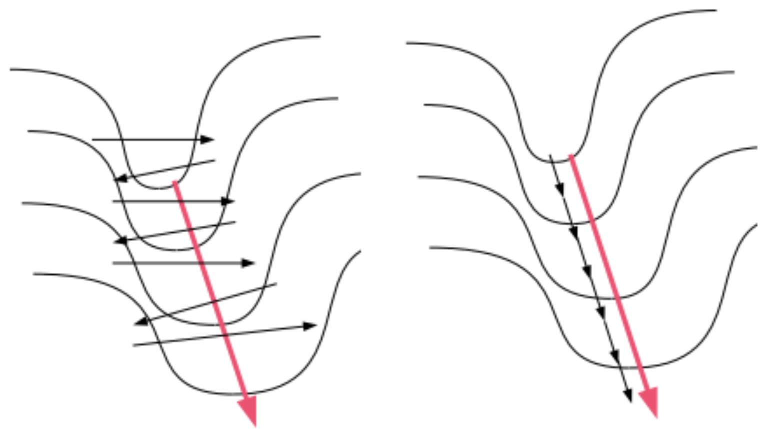


Issues (cont): Slow Convergence

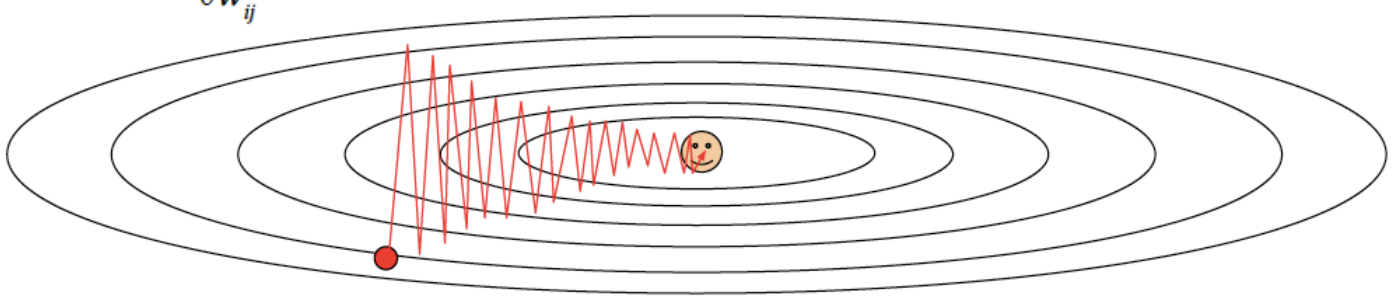
$L(w_1, w_2)$

$\frac{\partial L}{\partial w_1}$ Large

$\frac{\partial L}{\partial w_2}$ Small

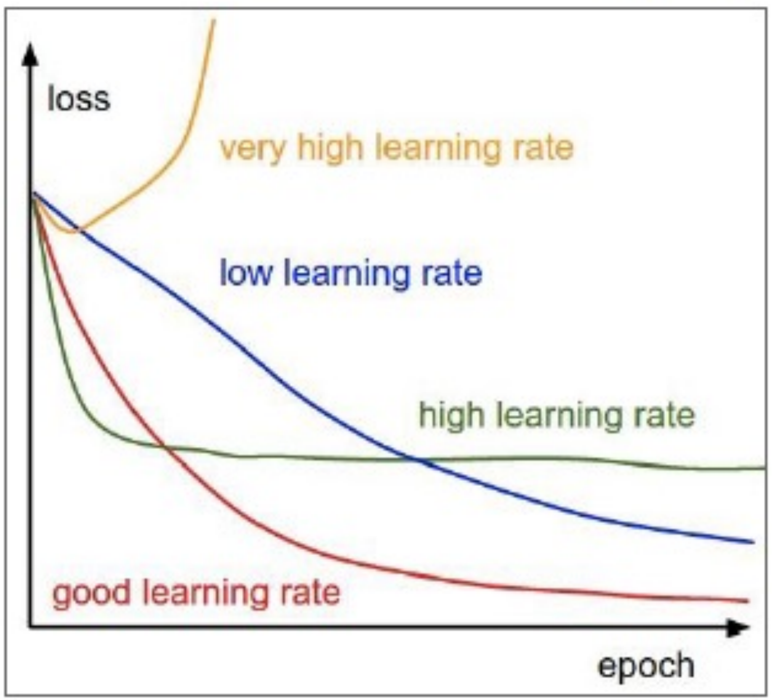


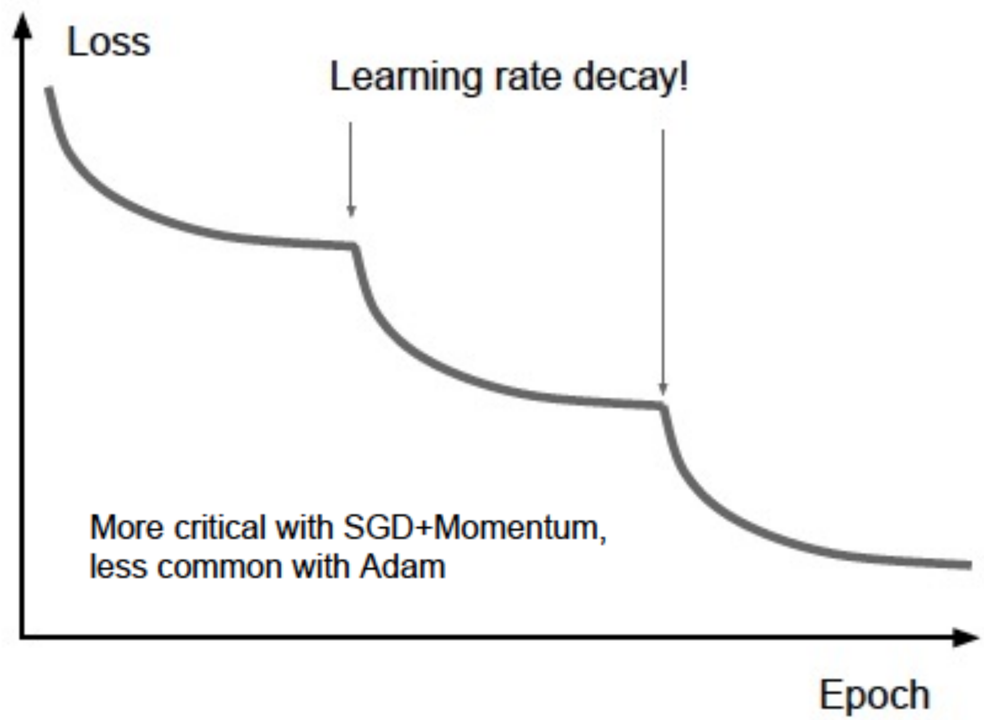
$$w_{ij}^{(r)} \leftarrow w_{ij}^{(r)} - \eta \frac{\partial L_m}{\partial w_{ij}^{(r)}}$$



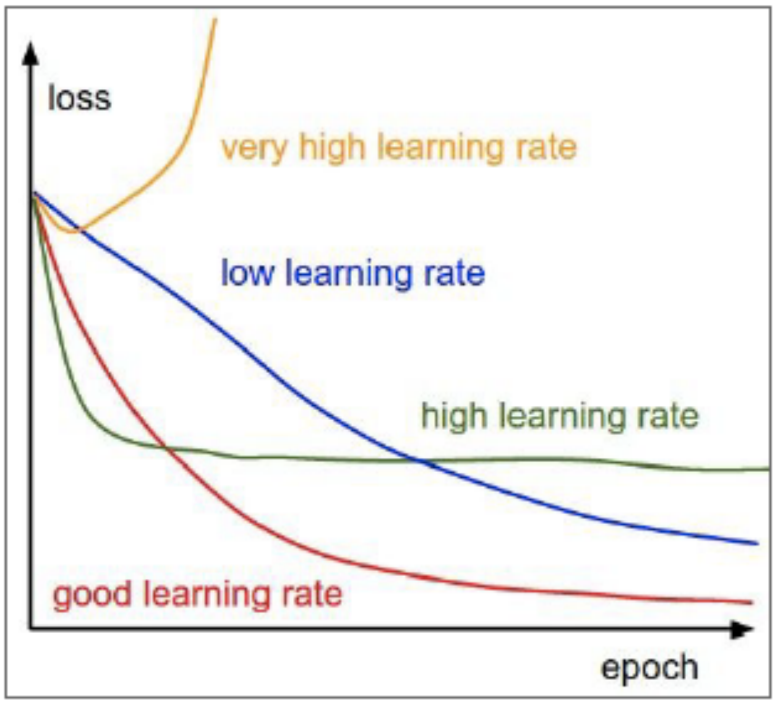
Loss changes quickly in one direction and slowly in another
Slow progress in shallow direction, Jitter along steep direction

Effect of Learning Rate on Optimization





Effect of Learning Rate on Optimization



=> Learning rate decay over time!

step decay:
e.g. decay learning rate by half every few epochs.

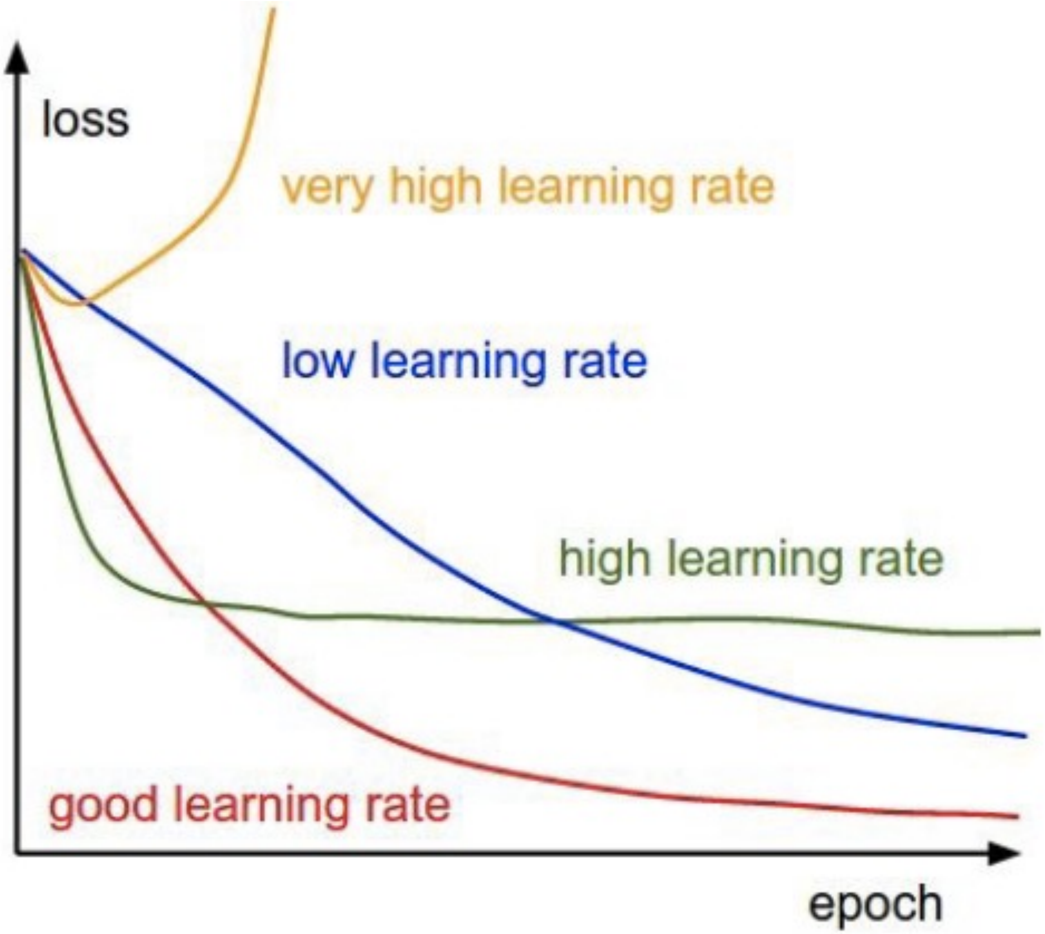
exponential decay:

$$\alpha = \alpha_0 e^{-kt}$$

1/t decay:

$$\alpha = \alpha_0 / (1 + kt)$$

Effect of Learning Rate η on Optimization



$$w_{ij} \leftarrow w_{ij} - \eta \frac{\partial \mathcal{L}}{\partial w_{ij}}$$

AdaGrad

$$w_{ij} \leftarrow w_{ij} - \eta \frac{\partial \mathcal{L}}{\partial w_{ij}}$$

- Builds Learning Rate Adaptation into SGD
- Every parameter gets its own Learning Rate

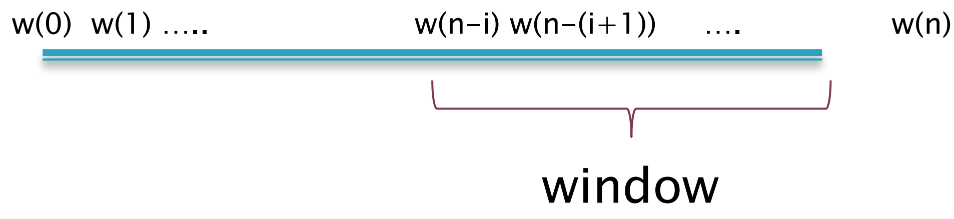
Update Rule

$$w(n+1) = w(n) - \eta \frac{\frac{\partial L(n)}{\partial w}}{\sqrt{\sum_{i=1}^{n} \left[\frac{\partial L(i)}{\partial w}\right]^2 + 10^{-7}}}$$

Benefits:

- Adapts update step on a per-direction basis, such that
  - Steep gradients lead to smaller updates
  - Shallow gradients lead to larger updates
- Updates decay over time – a desirable property, but can also be a problem

RMSProp



Update Rule

$$\Delta(n) = \alpha\Delta(n-1) + (1-\alpha)\left[\frac{\partial L(n)}{\partial w}\right]^2$$

$$w(n+1) = w(n) - \eta\frac{\frac{\partial L(n)}{\partial w}}{\sqrt{\Delta(n)}}$$

- Low Pass Filter has a windowing effect: Forgets gradients that are far back in time.
- Retains the benefits of ADAGRAD while avoiding the decay of the Learning Rate to zero.

Improved SGD

Learning Rate Adaptation

- AdaGrad
- RMSProp

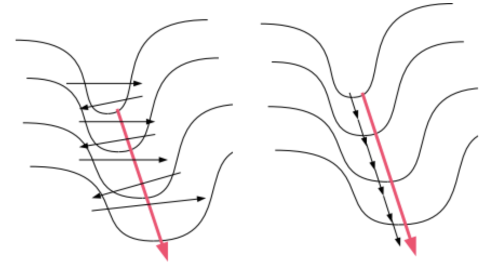
Faster Convergence

- Momentum
- Nesterov Momentum

Combination Technique

- Adam

Momentum



Main Idea: Accelerate progress along dimensions in which gradient consistently points in the same direction and slow progress along dimensions where the sign of the gradient continues to change

$$v(n) = \rho v(n-1) - \eta \frac{\partial \mathcal{L}(n)}{\partial w}$$

$$w(n+1) = w(n) + v(n)$$

New Parameter

ρ: Momentum Coefficient

$$\boxed{w(n+1) = w(n) - \eta \sum_{i=0}^{n} \rho^{n-i} \frac{\partial \mathcal{L}(i)}{\partial w}}$$

$$w(n+1) \leftarrow w(n) - \eta \frac{\partial \mathcal{L}}{\partial w}$$

Momentum (cont)

The ball accumulates momentum as it rolls downhill, becoming faster and faster along the way

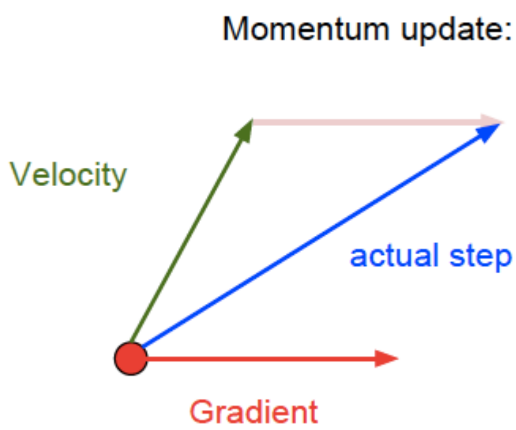
Helps with both Local Minima as well as Saddle Points

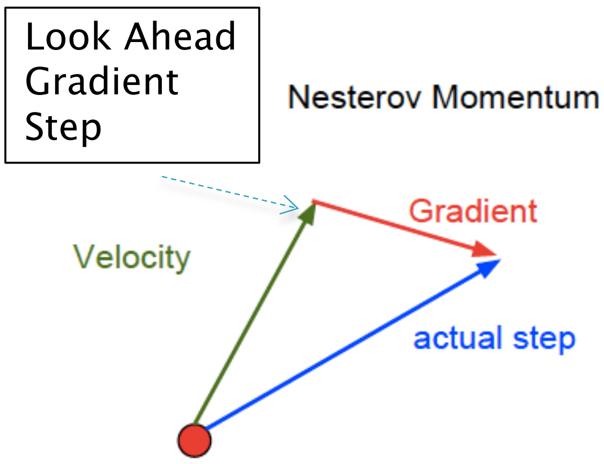
ρ: Momentum Co-efficient

$\rho = 0$: Defaults to SGD, only the latest gradient used -> No momentum

$\rho = 1$: All of the last n gradients used, large momentum

Nesterov Momentum





$$v(n) = \rho v(n-1) - \eta \frac{\partial \mathcal{L}(w(n))}{\partial w}$$

$$w(n+1) = w(n) + v(n)$$

$$v(n) = \rho v(n-1) - \eta \frac{\partial \mathcal{L}(w(n) + \rho v(n-1))}{\partial w}$$

$$w(n+1) = w(n) + v(n)$$

- Nesterov Momentum works better

Adam

Combines ADAGRAD with MOMENTUM

$$\Lambda(n) = \beta\Lambda(n-1) + (1-\beta)\frac{\partial L(n)}{\partial w}$$

Momentum Like

$$\Delta(n) = \alpha\Delta(n-1) + (1-\alpha)\left[\frac{\partial L(n)}{\partial w}\right]^2$$

ADAGRAD Like

$$w(n+1) = w(n) - \eta\frac{\Lambda(n)}{\sqrt{\Delta(n)+10^{-7}}}$$

Typical parameter values:

$\beta = 0.9$

$\alpha = 0.999$

$\eta = 10^{-3}$ *or* $5X10^{-4}$

$$\Lambda(n) \leftarrow \frac{\Lambda(n)}{1-\beta^T}$$

$$\Delta(n) \leftarrow \frac{\Delta(n)}{1-\alpha^T}$$

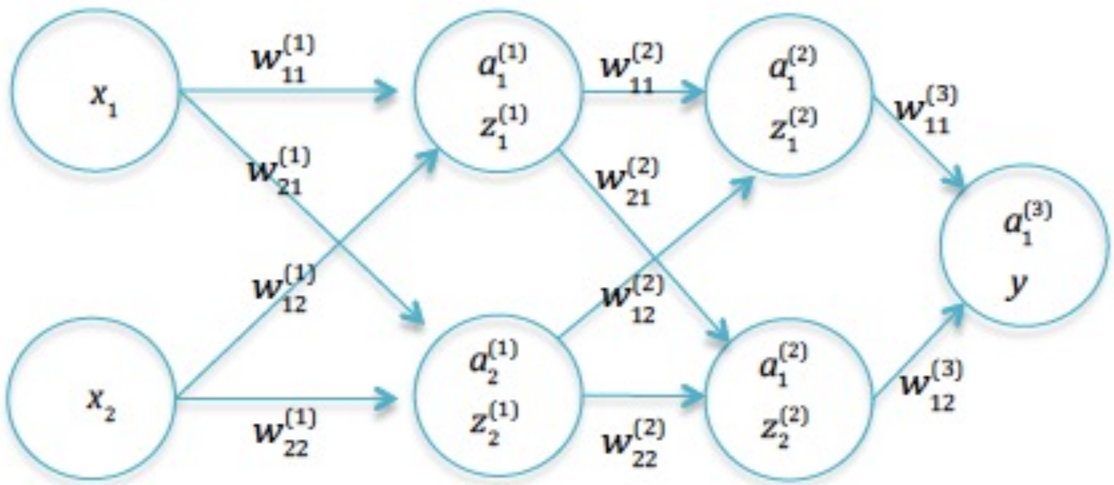
Optimizers in Keras

```
from keras import optimizers

model = Sequential()
model.add(Dense(64, kernel_initializer='uniform', input_shape=(10,)))
model.add(Activation('softmax'))

sgd = optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='mean_squared_error', optimizer=sgd)
```

Information available at:

https://keras.io/optimizers/

Example: Forward Pass



$$a_1^{(1)} = w_{11}^{(1)} x_1 + w_{12}^{(1)} x_2$$
$$z_1^{(1)} = f(a_1^{(1)})$$

$$a_2^{(1)} = w_{21}^{(1)} x_1 + w_{22}^{(1)} x_2$$
$$z_2^{(1)} = f(a_2^{(1)})$$

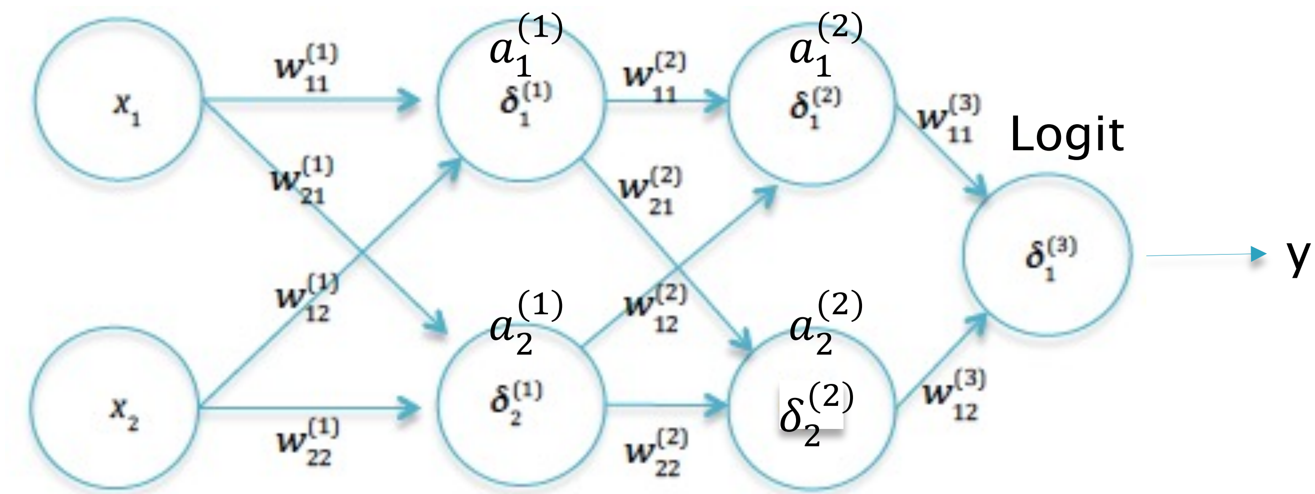
$$a_1^{(2)} = w_{11}^{(2)} z_1^{(1)} + w_{12}^{(2)} z_2^{(1)}$$
$$z_1^{(2)} = f(a_1^{(2)})$$

$$a_2^{(2)} = w_{21}^{(2)} z_1^{(1)} + w_{22}^{(2)} z_2^{(1)}$$
$$z_2^{(2)} = f(a_2^{(2)})$$

$$a_1^{(3)} = w_{11}^{(3)} z_1^{(2)} + w_{12}^{(3)} z_2^{(2)}$$
$$y = h(a_1^{(3)})$$

f: ReLu Function
h: Softmax Function

Example: Backward Pass



$$\delta_1^{(1)} = [w_{11}^{(2)}\delta_1^{(2)} + w_{21}^{(2)}\delta_2^{(2)}]f'(a_1^{(1)})$$

$$\delta_2^{(1)} = [w_{12}^{(2)}\delta_1^{(2)} + w_{22}^{(2)}\delta_2^{(2)}]f'(a_2^{(1)})$$

$$\delta_1^{(2)} = w_{11}^{(3)}\delta_1^{(3)} f'(a_1^{(2)})$$

$$\delta_2^{(2)} = w_{12}^{(3)}\delta_1^{(3)} f'(a_2^{(2)})$$

$$\delta_1^{(3)} = y - t$$

$$\Delta^{(3)} = \mathrm{Y} - \mathrm{T}$$

$$\Delta^{(1)} = f'\left(A^{(1)}\right) \odot \left(W^{(2)}\right)^T \Delta^{(2)}$$

$$\Delta^{(2)} = f'\left(A^{(2)}\right) \odot \left(W^{(3)}\right)^T \Delta^{(3)}$$

Further Reading

- Chapters 7: GradientDescentTechniques