Deep Learning Hands-On

Deep Learning in One Slide

- **What is it:**

Extract useful pattern from data

- **How:**

Neural Networks + optimization

- **Tools:**

Python + Tensorflow & Others

- **Hard Part:**

Good Questions + Good Data

Exciting Progress:

- Face recognition
- Image classification
- Speech recognition
- Text-to-speech generation
- Handwriting transcription
- Machine Translation
- Medical diagnosis
- Autonomous driving
- Game playing with deep RL

Deep Learning

Usual paradigm:

- we want to find a **model** that gives target **outputs** for particular **inputs**,
- we represent the model as a **function of parameters**,

$$f_\theta(x) = W_2\sigma\left(W_1 x + b_1\right) + b_2, \qquad \theta = \{W_1, W_2, b_1, b_2\}$$

- we can evaluate the model performance with a **differentiable loss function** that depends on **a set of data**,

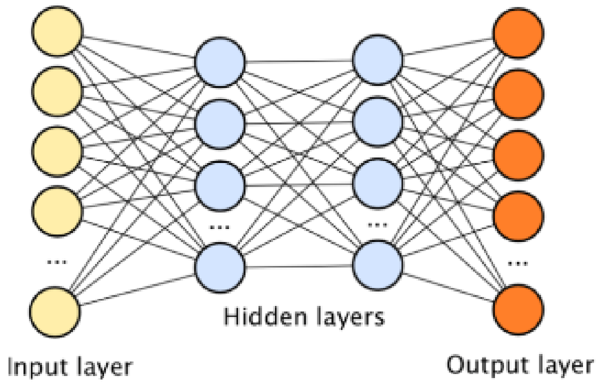
$$\mathcal{L}(\theta) = \frac{1}{|\mathcal{D}|} \sum_{(x,y)\in\mathcal{D}} L(x, y, f_\theta(x))$$

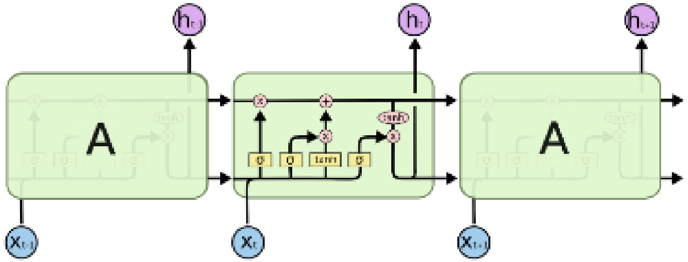
- and we find the optimal model by **gradient descent on the loss**.

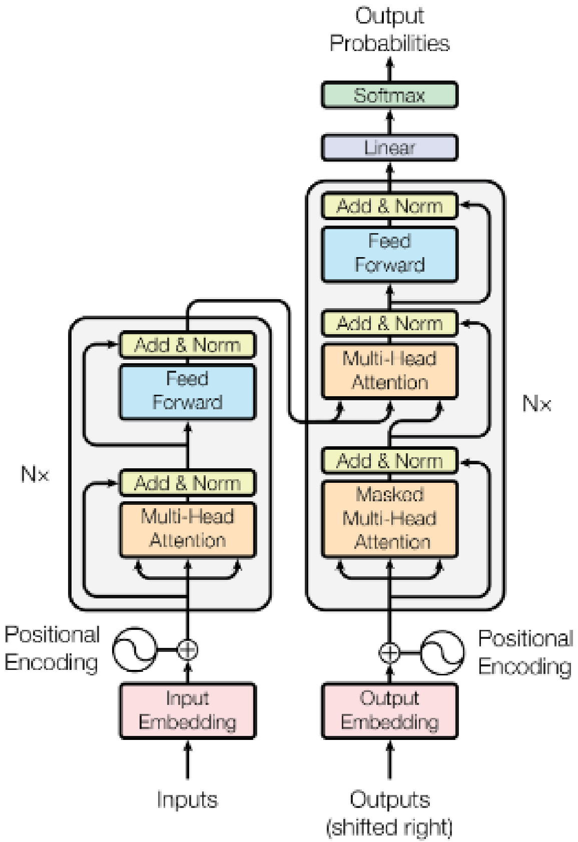
$$\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}(\theta)$$

Why Deep Learning?

- "Deep" refers to using **function composition** as the building block for models
- Deep models have many **layers**: output of one layer is input to next







Is all good?

- Regularizers make optimization problems better-behaved:

$$\mathcal{L}(\theta) \to \mathcal{L}(\theta) + \lambda\Omega(\theta)$$

- Normalization makes optimization easier:

$$a \to \frac{g}{\sigma}(a - \mu) + b$$

- Adaptive optimizers (eg Adam) makes optimization faster:

$$m \leftarrow \beta_1 m + (1 - \beta_1)g$$
$$v \leftarrow \beta_2 v + (1 - \beta_2)g^2$$
$$\theta \leftarrow \theta - \alpha \frac{m}{\sqrt{v} + \epsilon}$$

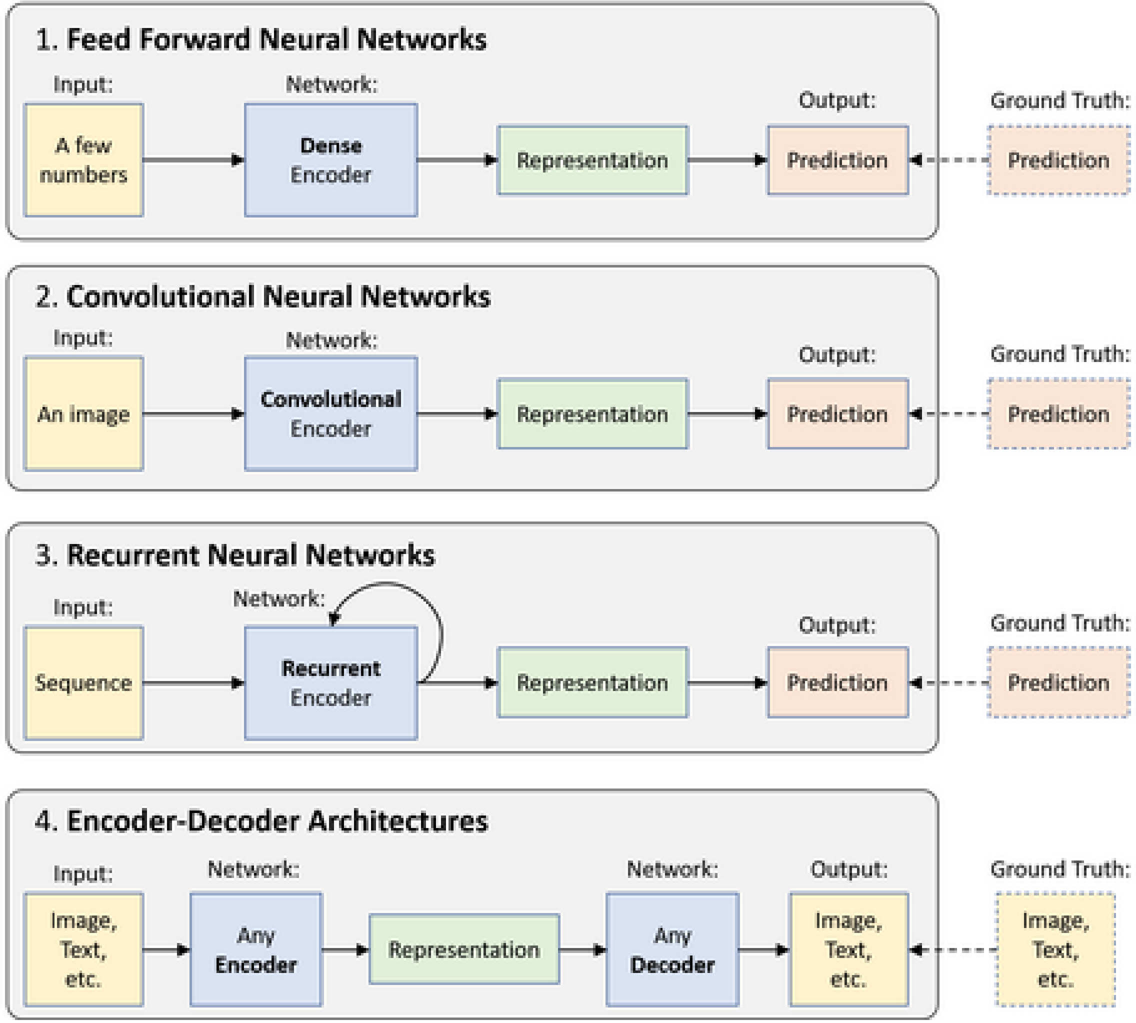
- Reparameterization trick comes in handy sometimes:

$$\mathop{\mathrm{E}}_{x \sim p_\theta}[F(x)] = \mathop{\mathrm{E}}_{z \sim \mathcal{N}}[F(\xi(\theta, z))]$$

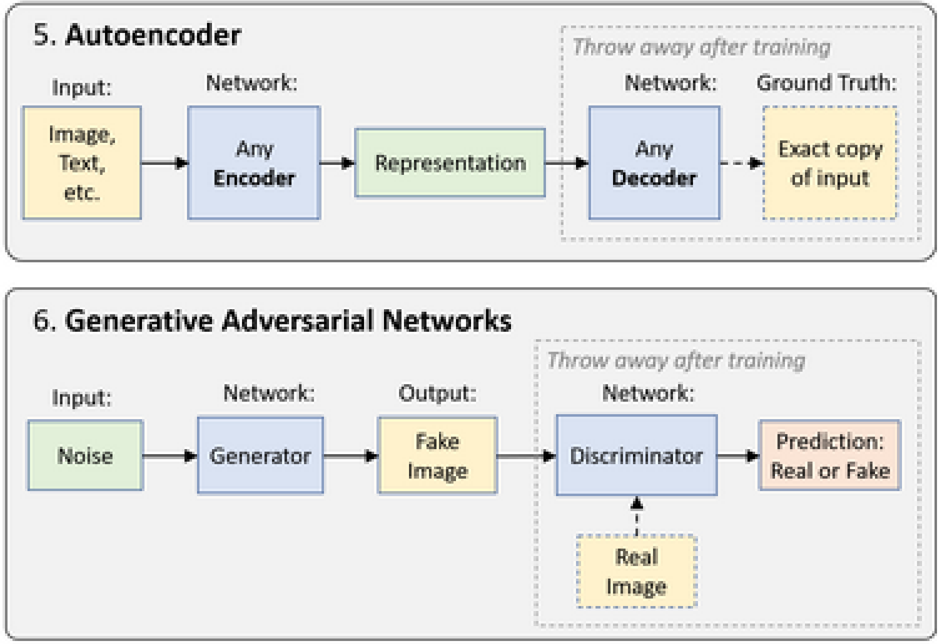
Tensorflow in One Slide

- **What is it:** Deep Learning Library
  - Open Source, Python, Google
- **Community:**
  - 117,000+ GitHub stars
  - Tensorflow.org: Blogs, Documentation, Youtube talks
- **Ecosystem:**
  - Keras: high-level API
  - Tensorflow.js: in the browser
  - Tensorflow Lite: on the phone
  - Colaboratory: in the cloud
  - TPU: optimized hardware
  - Tensorboard: visualization
  - Tensorflow Hub: graph models
- **Extras:**
  - Tensorflow Serving
  - Tensorflow Extended
  - Tensorflow Probability

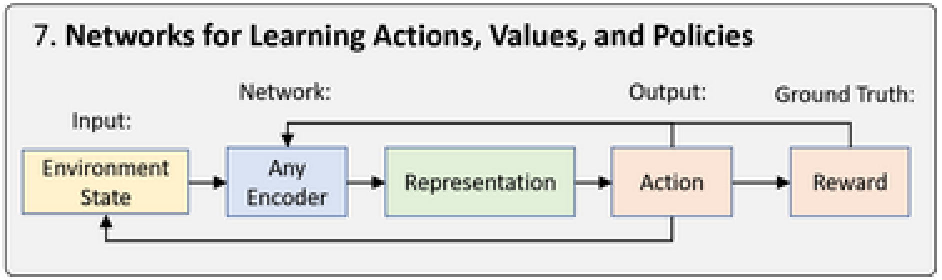
Supervised Learning



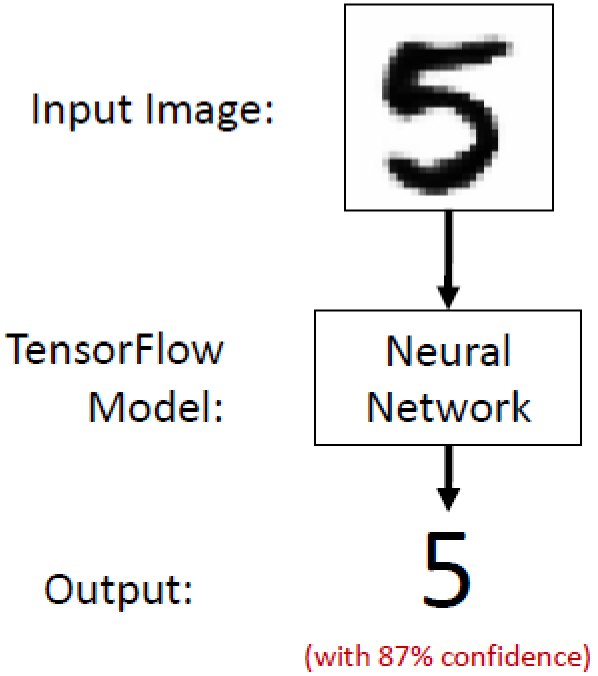
Unsupervised Learning



Reinforcement Learning

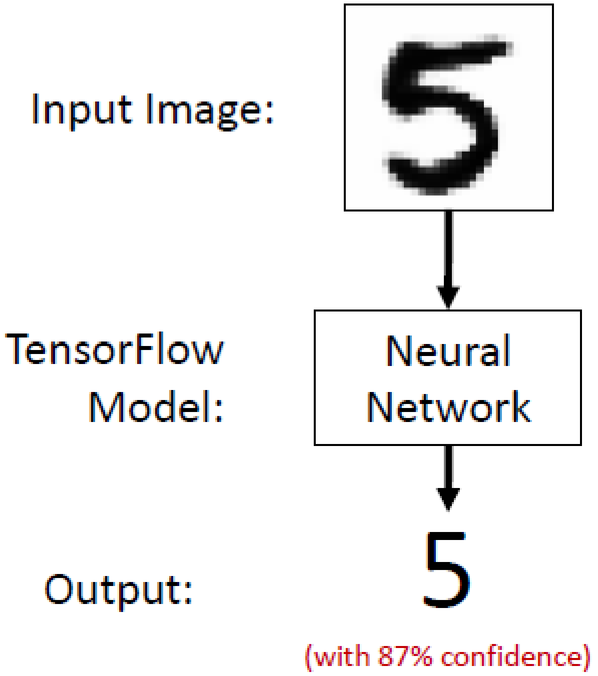


First Steps: **Small Example**



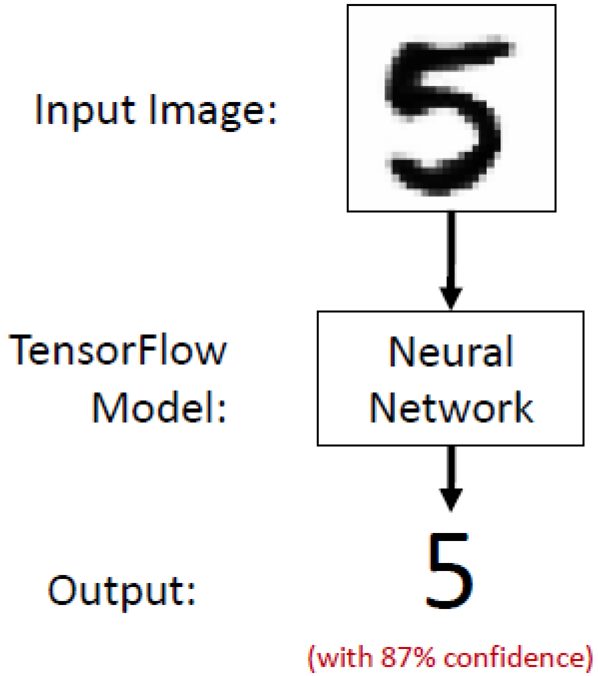

```python
# import tensorflow and mnist data
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)

# define placeholder for input, target, pred
x = tf.placeholder(tf.float32, [None, 784])

# Define the weight and bias variables
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))

# Define the output layer
y = tf.nn.softmax(tf.matmul(x, W) + b)
y_ = tf.placeholder(tf.float32, [None, 10])

# Cost function
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))

# Optimizer
train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)
init = tf.initialize_all_variables()
sess = tf.Session()
sess.run(init)

# Training
for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})

# Evaluate
correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
print(sess.run(accuracy, feed_dict={x: mnist.test.images, y_: mnist.test.labels}))
```

First Steps: **Small Example**



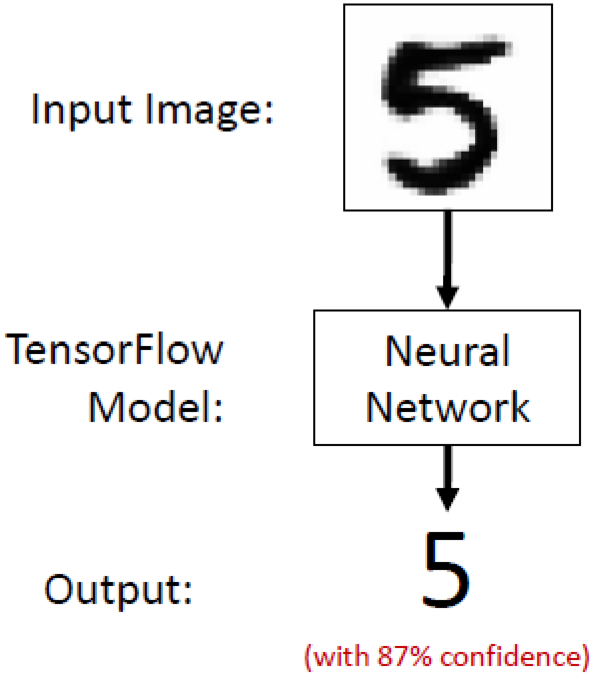

```python
# import tensorflow and mnist data
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)

# define placeholder for input, target, pred
x = tf.placeholder(tf.float32, [None, 784])

# Define the weight and bias variables
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))

# Define the output layer
y = tf.nn.softmax(tf.matmul(x, W) + b)
y_ = tf.placeholder(tf.float32, [None, 10])
```

First Steps: **Small Example**




```
# Cost function
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))

# Optimizer
train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)
init = tf.initialize_all_variables()
sess = tf.Session()
sess.run(init)
```

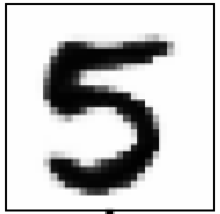
First Steps: **Small Example**




```
# Training
for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})

# Evaluate
correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
print(sess.run(accuracy, feed_dict={x: mnist.test.images, y_: mnist.test.labels}))
```

First Steps: **Small Example**

Input Image:



TensorFlow Model: Neural Network

Output: 5

(with 87% confidence)

1

```
# import tensorflow and keras (tf.keras not "vanilla" Keras)
import tensorflow as tf
from tensorflow import keras
```

2

```
# get data
(train_images, train_labels), (test_images, test_labels) = \
keras.datasets.mnist.load_data()
```

3

```
# setup model
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(128, activation=tf.nn.relu),
    keras.layers.Dense(10, activation=tf.nn.softmax)
])

model.compile(optimizer=tf.train.AdamOptimizer(),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

4

```
# train model
model.fit(train_images, train_labels, epochs=5)
```

5

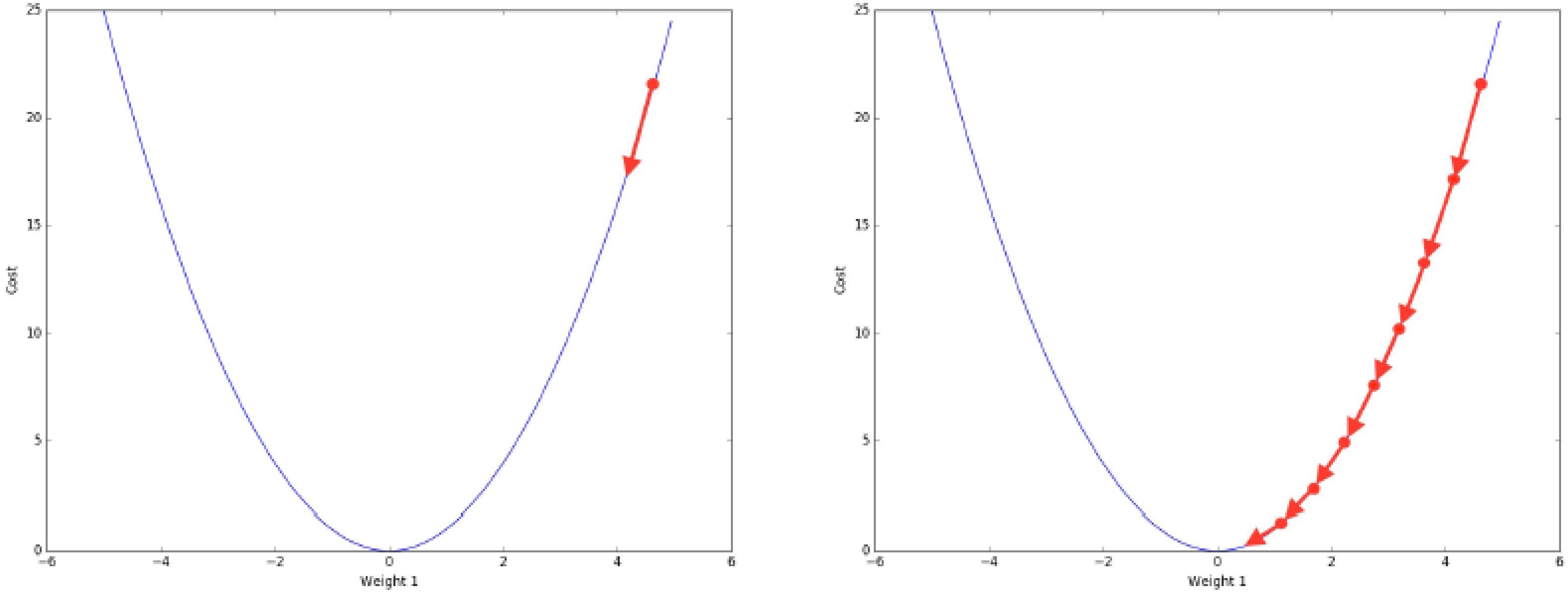
```
# evaluate
test_loss, test_acc = model.evaluate(test_images, test_labels)
print('test accuracy:', test_acc)
```

6

```
# make predictions
predictions = model.predict(test_images)
```

Learning is an Optimization Problem

Task: Update the **weights** and **biases** to decrease **loss function**



SGD: Stochastic Gradient Descent

Loss Functions

- **Quantifies gap between prediction and ground truth**

Mean Squared Error

Prediction

$$MSE = \frac{1}{N}\sum(t_i - s_i)^2$$

Ground Truth

- **Regression**

Cross Entropy Loss

Classes

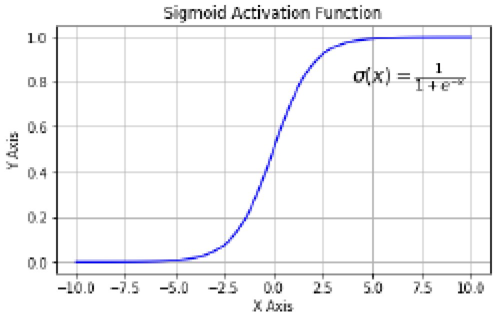
Prediction

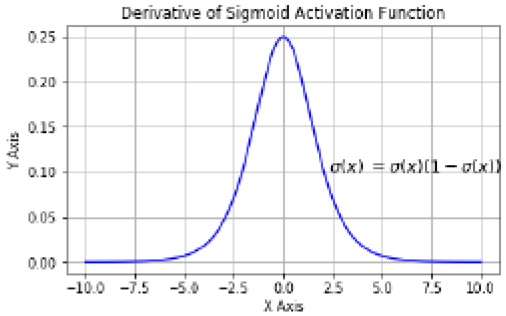
$$CE = -\sum_{i}^{C} t_i log(s_i)$$

Ground Truth {0,1}

- **Classification**

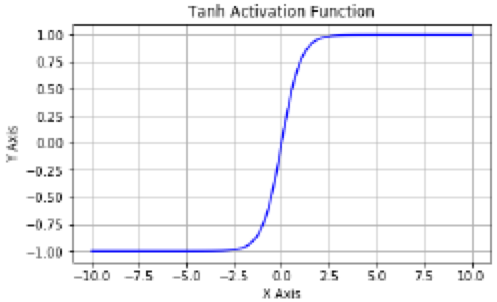
Activation Functions

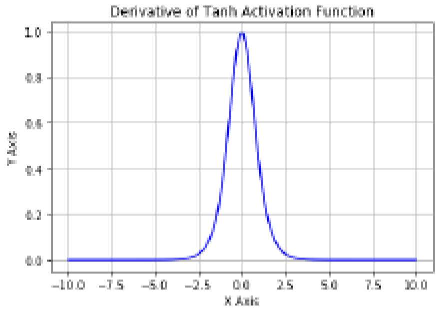




Sigmoid

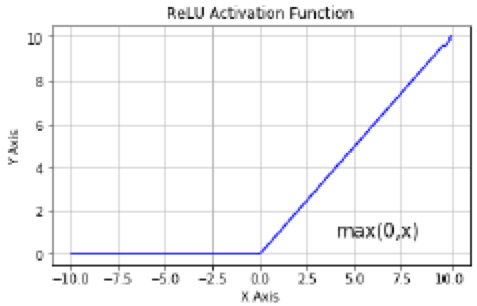
- Vanishing gradients
- Not zero centered

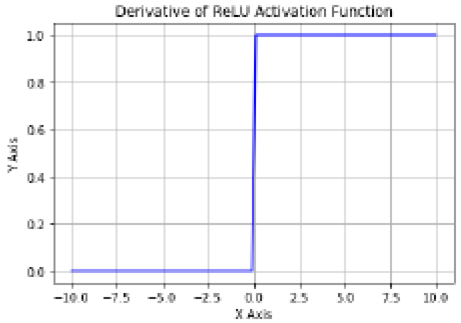




Tanh

- Vanishing gradients



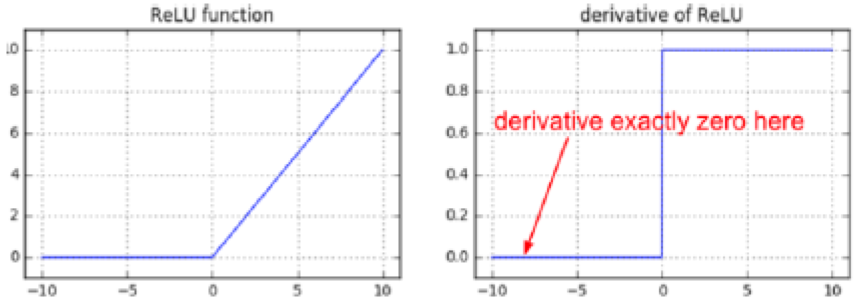


ReLU

- Not zero centered

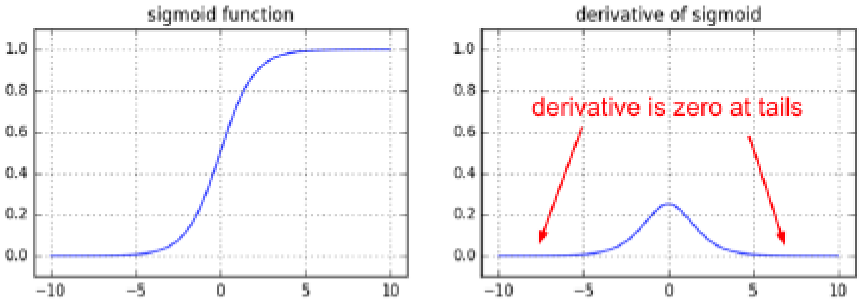
Common obstacles in DL

Dying ReLUs



- Poorly initialized, a neuron might never "fire" for the entire dataset
- Large parts of a network could be dead ReLUs!

Vanishing Gradients:

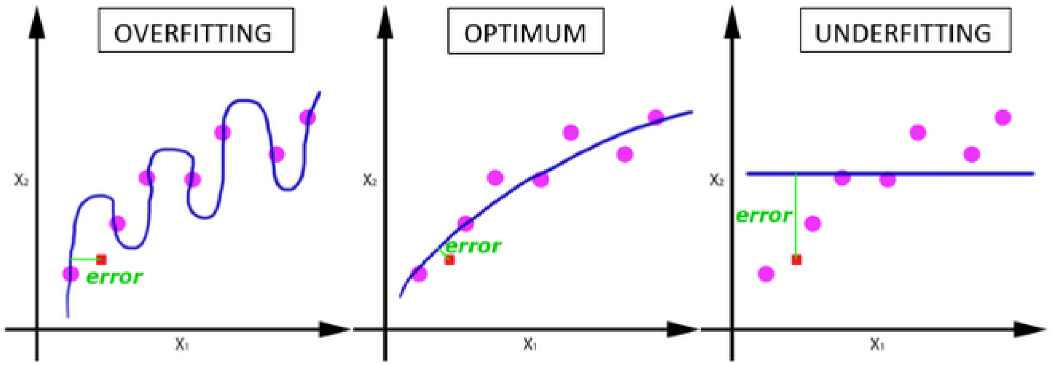


$$\frac{d\sigma(x)}{dx} = (1 - \sigma(x))\,\sigma(x)$$

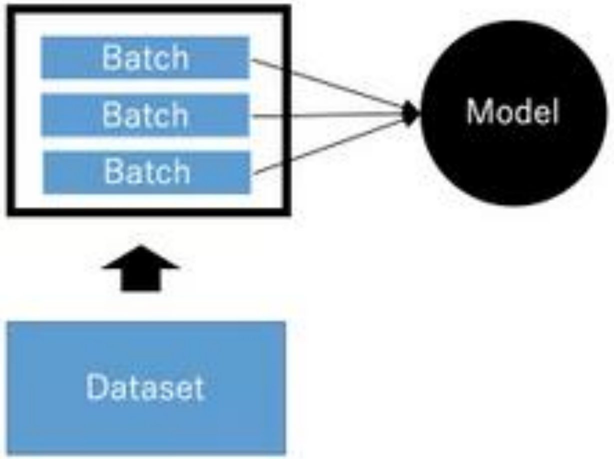
- **Backpropagation:**
  - Partial derivatives are small then the learning is slow

Overfitting and Regularization

- Network should be able to **generalize** to data it hasn't seen
- Big problem for **small dataset**

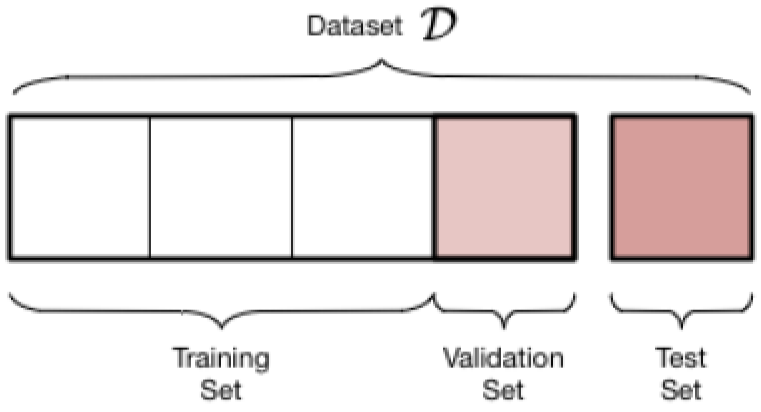


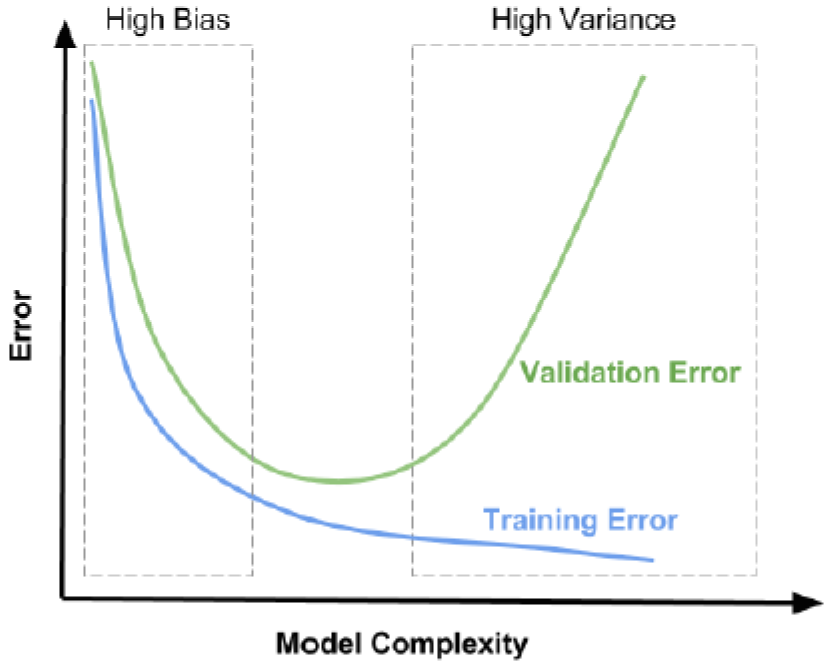
- **Mini-Batch Size:**
  - Large dataset cannot pass entire to the network, we should divide into a certain number of batches



Larger batch size = more speed
Smaller batch size = better generalization

Regularization: Early Stopping



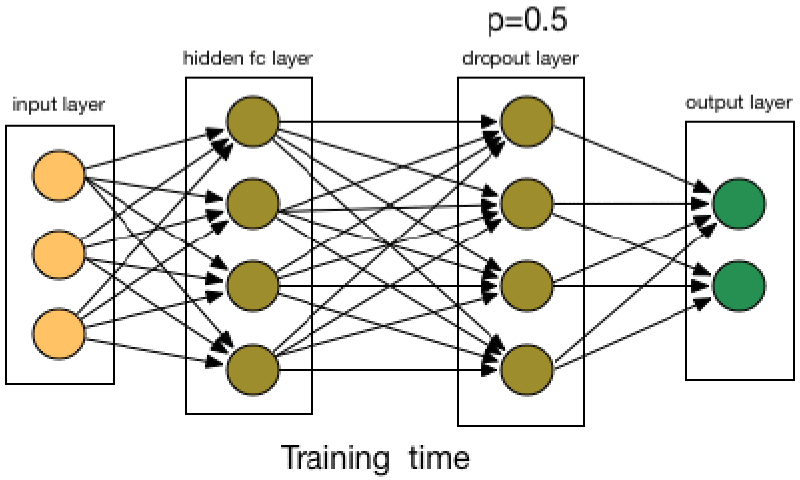


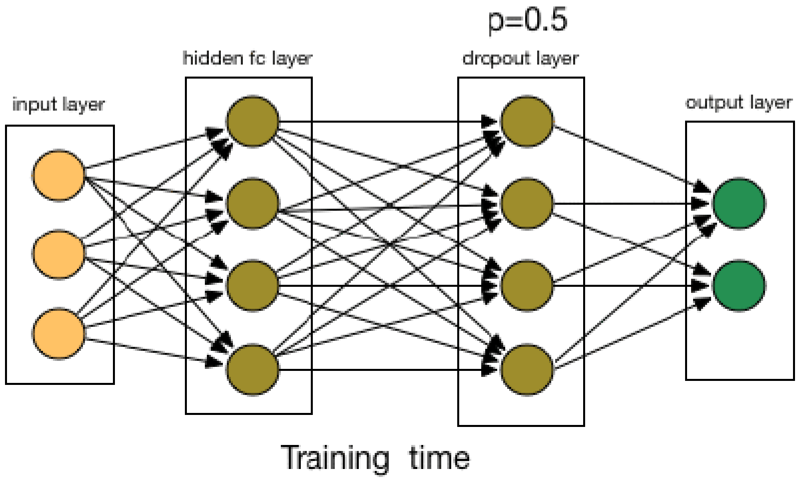
- Create a validation set (assume to be a representation of the testing set)
- **Early stopping:** Stop an save the last checkpoint when performance on validation set decreases

Regularization: Early Stopping

```
from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras.callbacks import EarlyStopping

# saves the model weights after each epoch if the validation loss decreased
checkpointer = ModelCheckpoint(filepath='./weights.hdf5', verbose=1, save_best_only=True)
early_stop   = EarlyStopping(monitor='val_loss', min_delta=0.01, patience=2, verbose=1, mode='auto')

model.fit(x = train_images,
          y = train_labels,
         verbose = 2,
         epochs = 5,
         batch_size = 32,
         validation_data = (test_images, test_labels),
         callbacks=[checkpointer, early_stop])
```

Regularization: Dropout





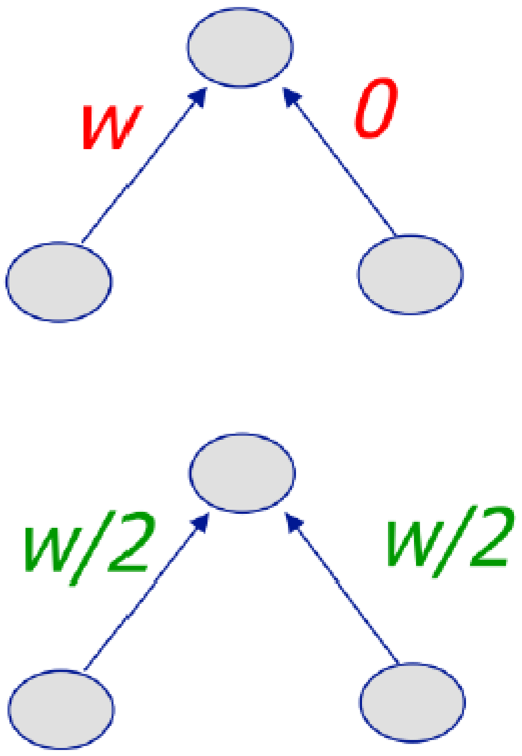
- **Dropout:** Randomly remove some nodes in the neural network (with the corresponding inputs and outputs)

- With a probability of keeping a node $p \geq 0.5$
- Dropout in the input layer should be higher *(can be replaced by noise)*
- Most deep learning frameworks have this function implemented

Regularization: Dropout

```python
rate = 0.5

model = K.Sequential([
    K.layers.Flatten(input_shape=(28,28)),
    K.layers.Dropout(rate),
    K.layers.Dense(128, activation=tf.nn.relu),
    K.layers.Dropout(rate),
    K.layers.Dense(10, activation=tf.nn.softmax)
])
print(model.summary())

model.compile( optimizer='adam',
               loss='sparse_categorical_crossentropy',
               metrics=['accuracy']
             )

model.fit(  x = train_images,
            y = train_labels,
            verbose = 2,
            epochs = 5,
            batch_size = 32,
            validation_data = (test_images, test_labels))
```

Regularization: Weight Penalty



- **L2 Penalty**
  - Keeps weight small unless error derivative is very large
  - Prevent from fitting sampling error
  - Output changes slower as the input changes
  - Balance similar inputs instead of weight all on one

- **L1 Penalty**
  - Allow for few weights to remain large
  - Penalize absolute weights

Regularization: Weight Penalty

```
from tensorflow.keras import regularizers
import tensorflow.keras.backend as bk

def l1_reg(weight_matrix):
    return 0.01 * bk.sum(bk.abs(weight_matrix))

model = K.Sequential()
model.add(K.layers.Flatten(input_shape=(28,28)))
model.add(K.layers.Dense(128, kernel_initializer='normal',
                               kernel_regularizer=regularizers.l2(0.01)))
model.add(K.layers.Activation('relu'))
model.add(K.layers.Dropout(rate))
model.add(K.layers.Dense(10,  kernel_initializer='uniform',
                               kernel_regularizer=l1_reg))
model.add(K.layers.Activation('softmax'))

print(model.summary())
```

Regularization: Normalization

- **Input Normalization**
  - Manage the input scale
  - Whitening
  - Scale according to the mean and std.

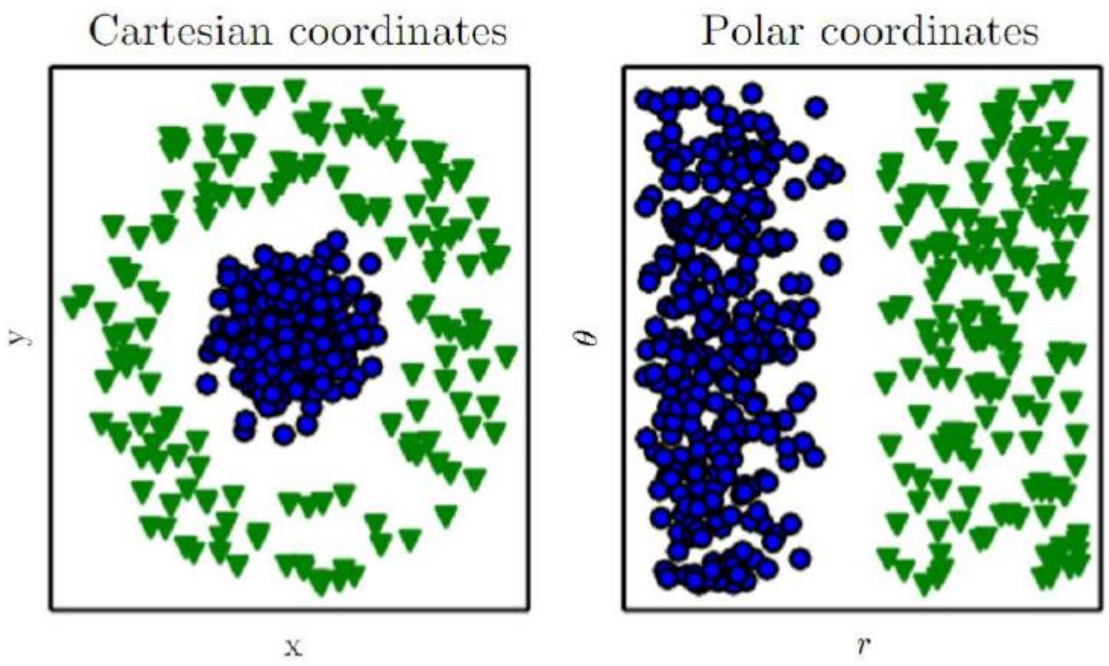
- **Batch Normalization**
  - Normalize hidden layer inputs
  - Use mini-batch mean and variance
  - Reduce the impact of earlier layers to more deep layers

- **Batch Renormalization**
  - Fixes difference b/w training and inference by keeping a moving average asymptotically approaching a global normalization

Regularization: Normalization

```
model = K.Sequential()
model.add(K.layers.Flatten(input_shape=(28,28)))
model.add(K.layers.Dense(128, kernel_initializer='normal'))
model.add(K.layers.Activation('relu'))
model.add(K.layers.BatchNormalization())
model.add(K.layers.Dense(10, kernel_initializer='normal'))
model.add(K.layers.Activation('softmax'))

print(model.summary())

model.compile(optimizer=tf.train.AdamOptimizer(learning_rate=0.003),
            loss='sparse_categorical_crossentropy',
             metrics=['accuracy']
            )

model.fit(  x = train_images,
            y = train_labels,
            verbose = 2,
            epochs = 5,
            batch_size = 32,
            validation_data = (test_images, test_labels))
```

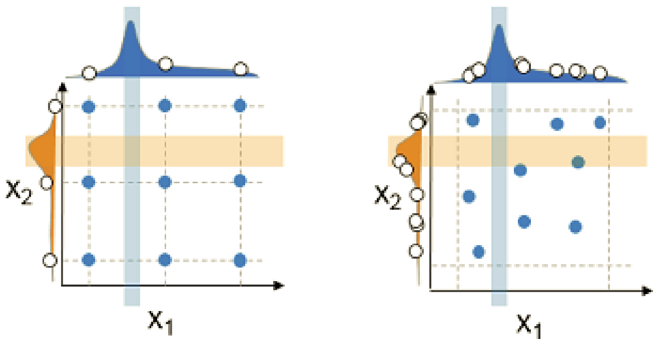
Representation and Data Dimension



	very similar dataset	very different dataset
very little data	Use Linear Classifier on top layer	You're in trouble… Try linear classifier from different stages
quite a lot of data	Finetune a few layers	Finetune a larger number of layers

- **Optimization**
  - Learning rate
  - Batch size
  - Hidden nodes

Hyper-parameters can be tuned by an Grid Search



Standard Grid Search

Random Search

Software

Keras

TensorFlow

- High level API, run on top
- Of TensorFlow, Theano
- Run CPU-GPU

- Grid Search and Random Search.
- Choose Model
- Parallelism
- Cross-validation between models

scikit learn

python

- Tools for data mining
- Numpy, SciPy, Matplotlib

Tensorflow Model → Keras Abstraction → Grid Search Sklearn → Obtain best Parameters

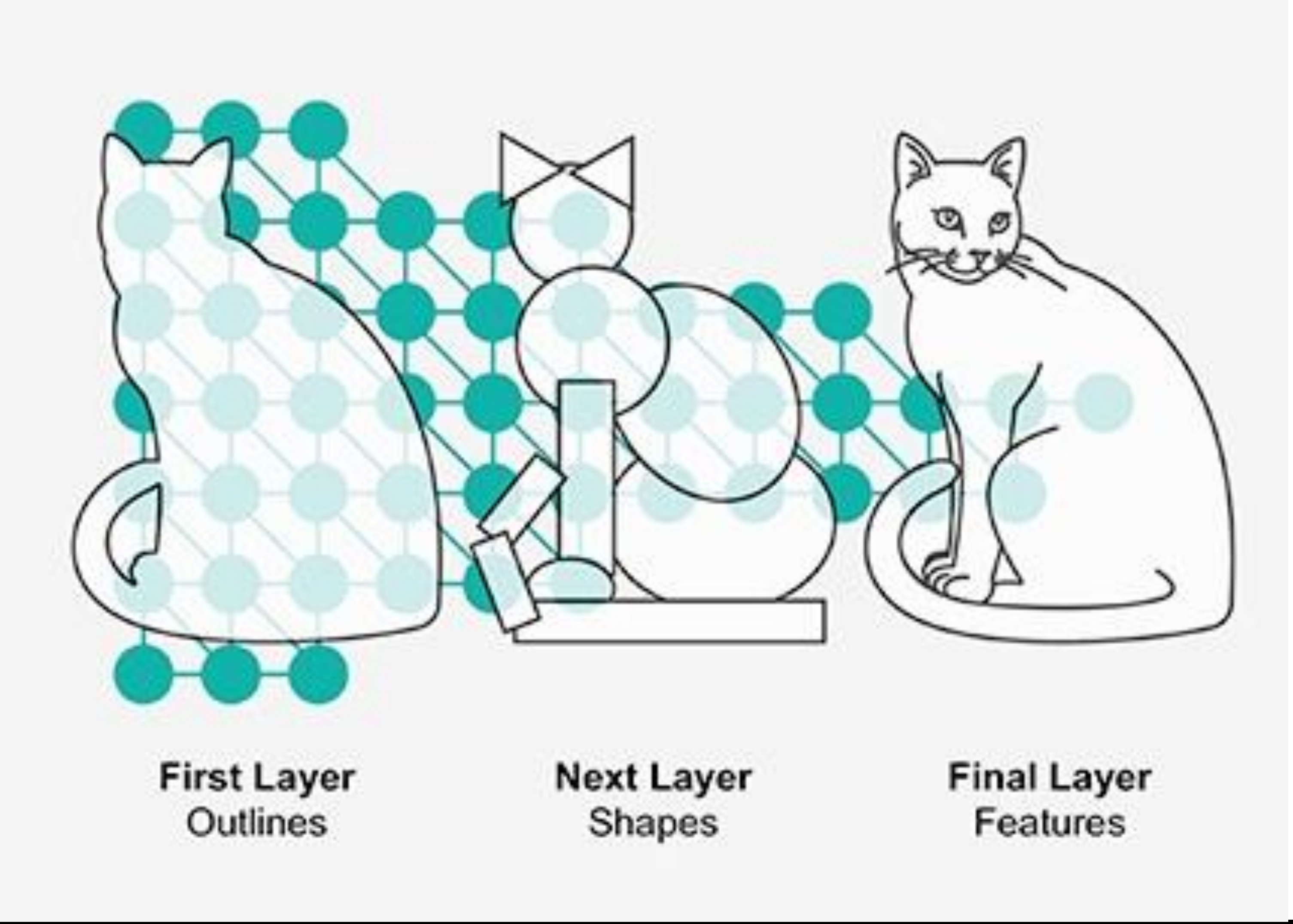
Grid-Search

```
# Use scikit-learn to grid search the batch size and epochs
import numpy
from sklearn.model_selection import GridSearchCV
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier
# Function to create model, required for KerasClassifier
def create_model():
    # create model
    model = K.Sequential()
    model.add(K.layers.Flatten(input_shape=(28,28)))
    model.add(K.layers.Dense(128, activation='relu'))
    model.add(K.layers.Dense(10, activation='softmax'))
    # Compile model
    model.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model

# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
```

Grid-Search

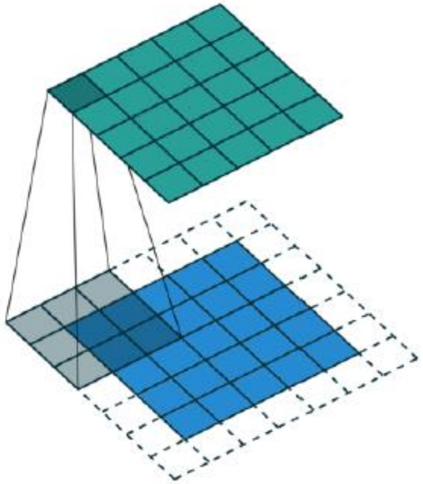
```
# create model
model = KerasClassifier(build_fn=create_model, verbose=0)
# define the grid search parameters
batch_size = [16, 32]
epochs = [1, 3, 5]
param_grid = dict(batch_size=batch_size, epochs=epochs)
grid = GridSearchCV(estimator=model, param_grid=param_grid, cv=3, n_jobs=1)
grid_result = grid.fit(test_images, test_labels)


# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

BREAK

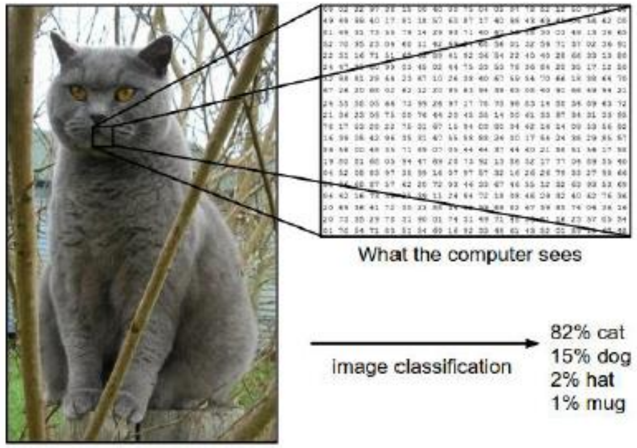


Convolutional Neural Networks

Convolution Networks



Filter that take
Advantage of
Spatial invariance



Convolution Networks

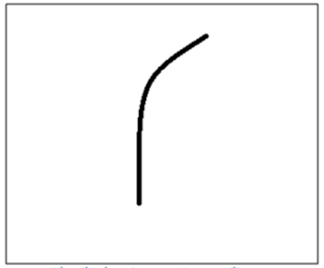
Input Data

Original image

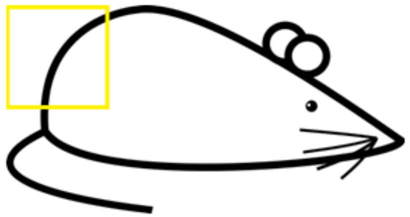
Convolution filter

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter



Visualization of a curve detector filter



Visualization of the filter on the image

0	0	0	0	0	0	30
0	0	0	0	50	50	50
0	0	0	20	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0

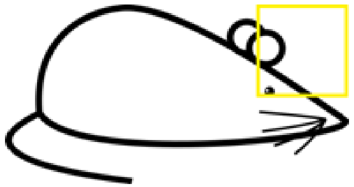
Pixel representation of the receptive field

*

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter

Multiplication and Summation = (50*30)+(50*30)+(50*30)+(20*30)+(50*30) = 6600 (A large number!)



Visualization of the filter on the image

0	0	0	0	0	0	0
0	40	0	0	0	0	0
40	0	40	0	0	0	0
40	20	0	0	0	0	0
0	50	0	0	0	0	0
0	0	50	0	0	0	0
25	25	0	50	0	0	0

Pixel representation of receptive field

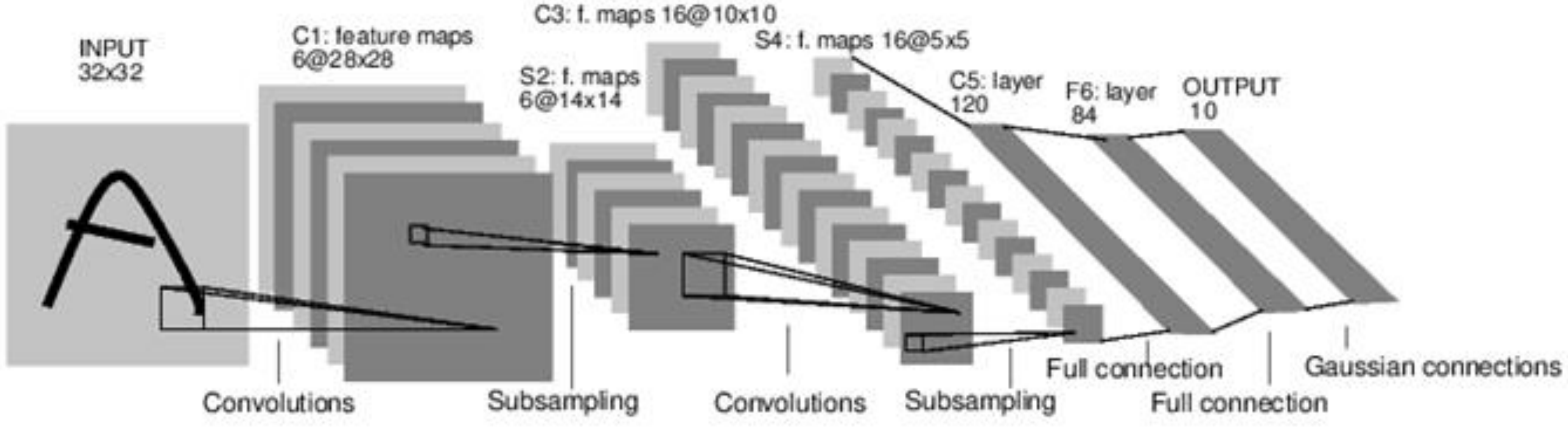
*

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter

Multiplication and Summation = 0

Convolution Networks

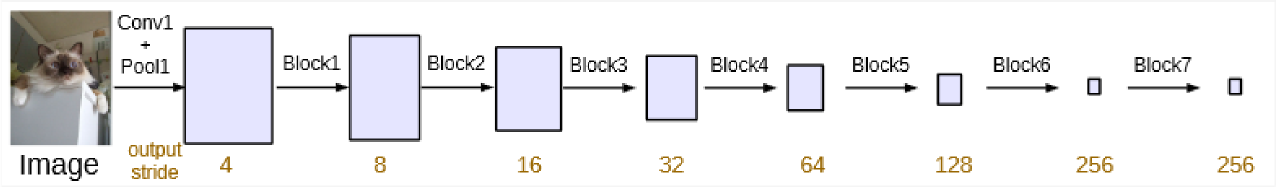


A Full Convolutional Neural Network (LeNet)

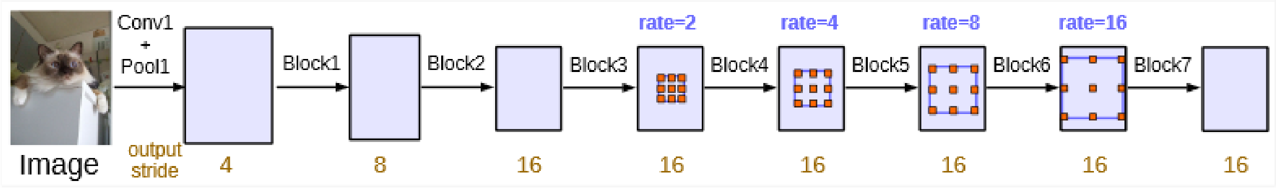
CNN

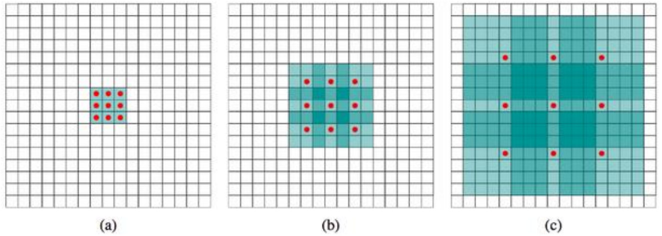
```
# define the input
xi = K.Input((28,28,1))

# define the network
xn = K.layers.Dropout(rate)(xi)
xn = K.layers.Conv2D(64, (3,3), padding='same', activation='relu')(xn)
xn = K.layers.BatchNormalization()(xn)
xn = K.layers.Conv2D( 1, (3,3), padding='same', activation='relu')(xn)
xn = K.layers.Flatten()(xn)

# define the output
xo = K.layers.Dense(10, activation='softmax')(xn)

model = K.Model(inputs=[xi], outputs=[xo])
print(model.summary())
```

Atrous Convolution



(a) Going deeper without atrous convolution.





- Remove the down-sampling from the last pooling layers.
- Up-sample the original filter by a factor of the strides:

  Atrous convolution for 1-D signal:

$$y[i] = \sum_{k=1}^{K} x[i + r \cdot k] w[k]$$

Introduce zeros between filter values

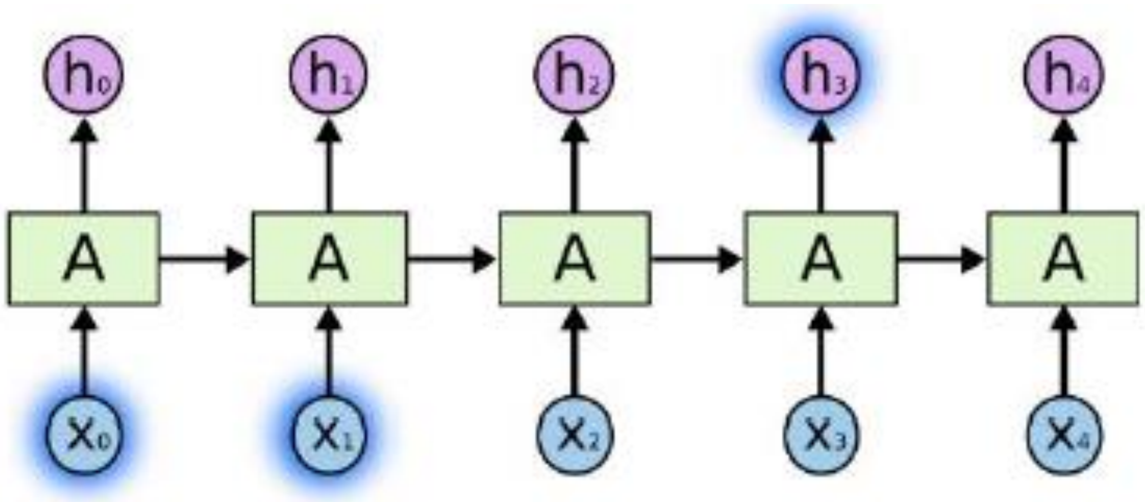
$x[i]$ 1-D input signal

$w[k]$ filter of length K

r *rate* parameter corresponds to the stride with which we sample the input signal.

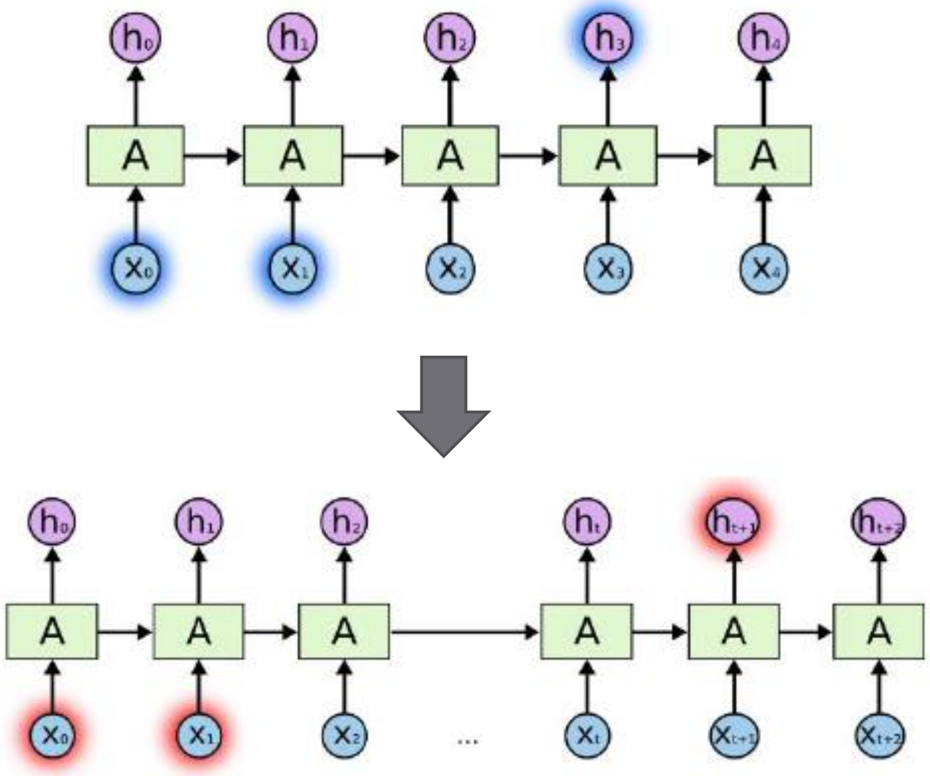
$y[i]$ output of atrous convolution.

- Note: standard convolution is a special case for *rate* $r=1$.



Recurrent Neural Networks

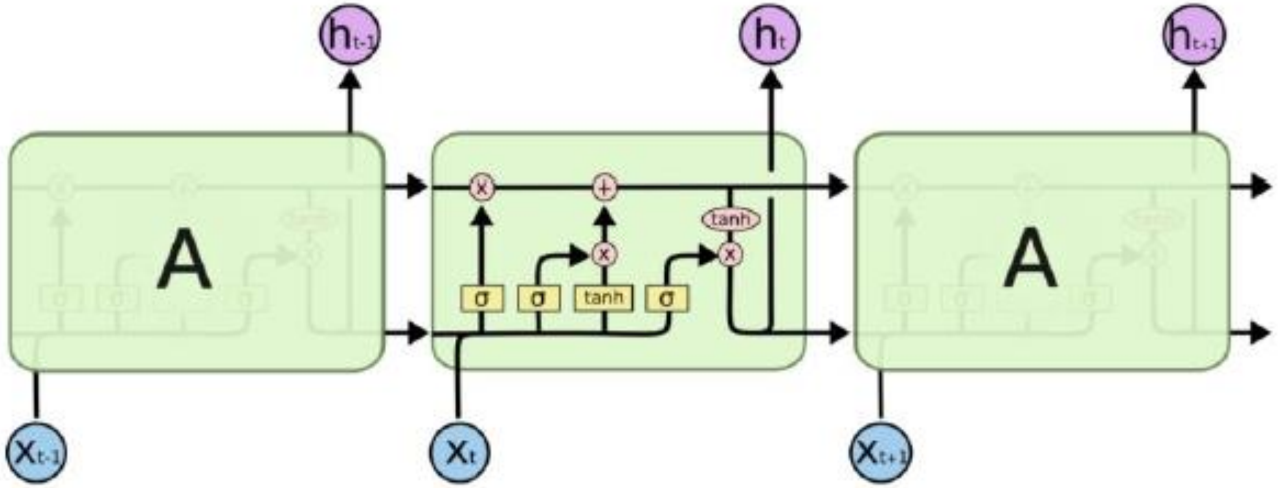
Recurrent Neural Networks
(Long-Term Dependency)



- **Short-Tem dependence**
  - Bob is eating an apple.
- **Long-Term dependence**
  - Bob likes apples. He is hungry and decided to have a snack. So now he is eating an apple.

In theory, vanilla RNN can handle arbitrarily long-term dependence
(**In practice**, it's difficult)

Long Short-Term Memory (LSTM)



- Pipeline for storing a **previous state** and process **new incoming data**:
  - Decide what to forget
  - Decide what to remember
  - Decide what to output

- ***RNN architectures***
  - Long Short Term Memory - LSTM

Useful

Training a Gradient Descent Algorithm

- Vanishing Gradient
- Exploding Gradient

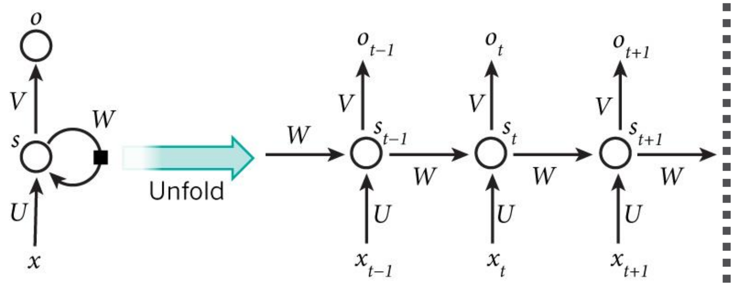
Includes new terms:

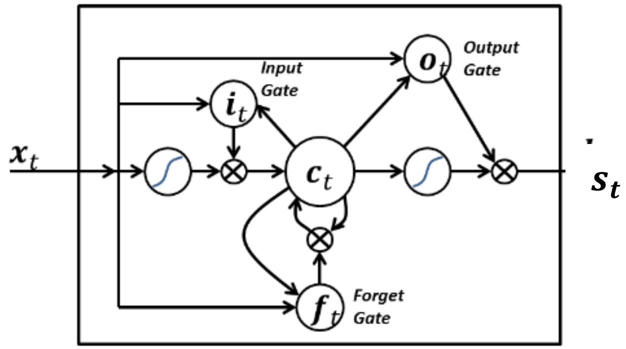
- Input gate
- Output gate
  - Effect state of memory
- Recurrent connection
- Forget gate
  - Module the memory cell self-recurrent connection
- State of cell
  - Remember or forget

Sepp Hochreiter; Jürgen Schmidhuber (1997). "Long short-term memory"

K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink and J. Schmidhuber, "LSTM: A Search Space Odyssey,"

Diagram





$$D_i = [A_i, G_i] = [batch, timestep, depth]$$

$$D_i = [A_i, G_i] = [timestep, (batch, depth)]$$

Equations

LSTM

Input gate → Incoming signal alter state of memory

$$i_t = \sigma(U_i x_t + W_i s_{t-1} + b_i)$$

Candidate Cell → States of memory at time t

$$\widetilde{c_t} = tanh(U_c x_t + W_c s_{t-1} + b_c)$$

Forget gate →activation of memory

$$f_t = \sigma\left(U_f x_t + W_f s_{t-1} + b_f\right)$$

State of cell → new state

$$c_t = f_t * c_{t-1} + i_t * \widetilde{c_t}$$

Output and hidden gate

$$o_t = \sigma(U_o x_t + W_o s_{t-1} + V_o c_t + b_o)$$

$$s_t = o_t * tanh(c_t)$$

LSTM

```
# define the input
xi = K.Input((28,28))

# define the network
xn = K.layers.Dropout(rate)(xi)
xn = K.layers.LSTM( 64, return_sequences=True)(xn)
xn = K.layers.BatchNormalization()(xn)
xn = K.layers.LSTM(64, return_sequences=False)(xn)

# define the output
xo = K.layers.Dense(10, activation='softmax')(xn)

model = K.Model(inputs=[xi], outputs=[xo])
print(model.summary())
```

- ***RNN architectures***
  - Gate Recurrent Unit

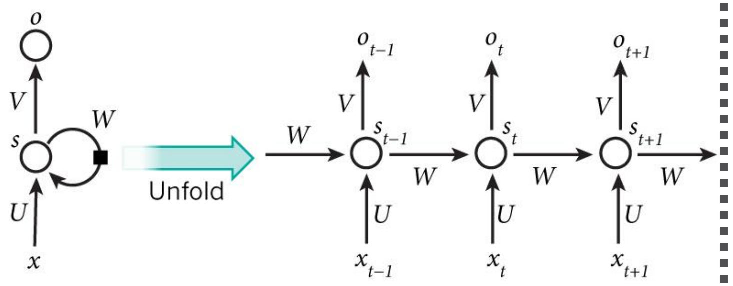
Useful

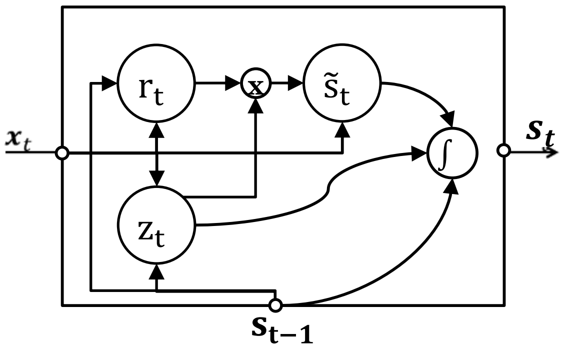
- It combines the forget and input gates into a single "update gate."
- It also merges the cell state and hidden state
- They have fewer parameters than LSTM
- Prove to be better for a specific cases (most of studies are in NLP or Music Generation)

Cho, Van Merrienboer, Bengio, "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation", 2014

Chung, Gulcehre, Cho, Bengio, "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling", 2014

Diagram





Equations

GRU

Update gate → Incoming signal alter state of memory

$$z_t = \sigma(U_i x_t + W_i s_{t-1} + b_z)$$

Reset gate → activation of memory

$$r_t = \sigma(U_r x_t + W_r s_{t-1} + b_r)$$

Candidate cell → States of memory

$$\tilde{s}_t = tanh(U_c x_t + W_c(r_t * s_{t-1}) + b_c)$$

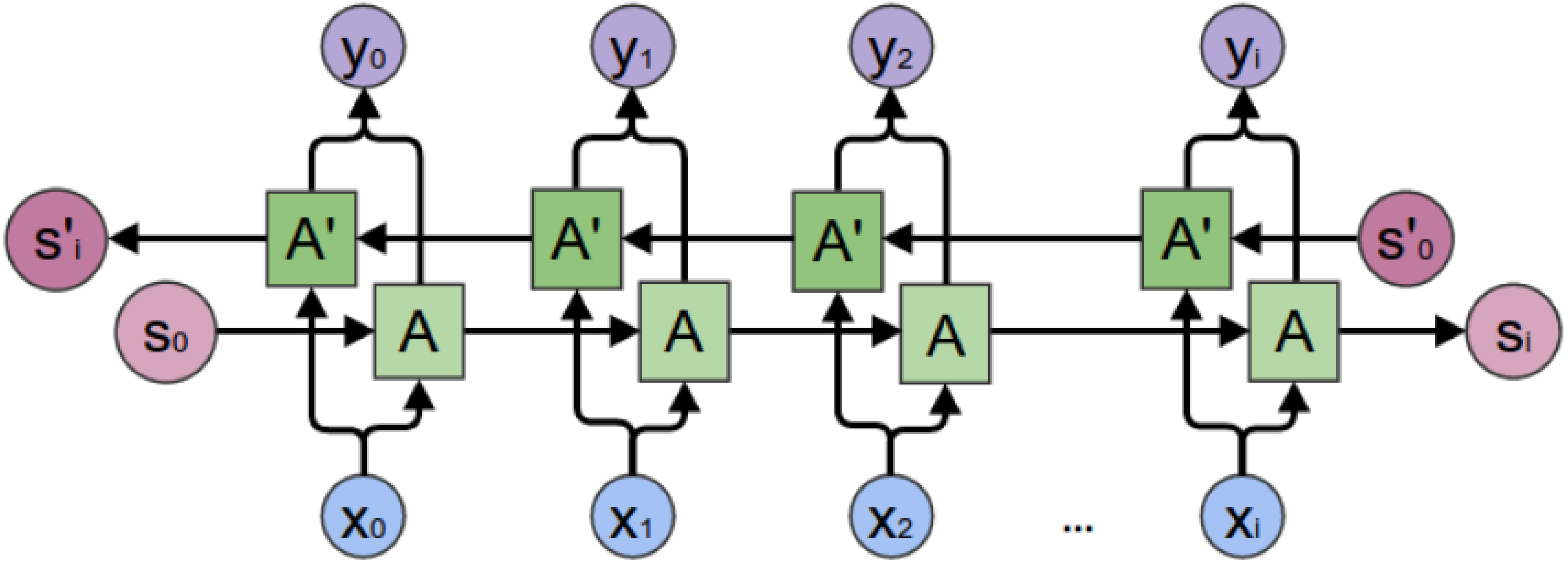
Output and hidden state

$$s_t = z_t * s_{t-1} + (1 - z_t) * \tilde{s}_t$$

GRU

```
# define the input
xi = K.Input((28,28))

# define the network
xn = K.layers.Dropout(rate)(xi)
xn = K.layers.GRU( 64, return_sequences=True)(xn)
xn = K.layers.BatchNormalization()(xn)
xn = K.layers.GRU(64, return_sequences=False)(xn)

# define the output
xo = K.layers.Dense(10, activation='softmax')(xn)

model = K.Model(inputs=[xi], outputs=[xo])
print(model.summary())
```

Bidirectional RNN



- Learn a representation after watching a sequence from both: **Previous** time steps and **future** time steps

- ***RNN architectures***
  - Bi-directional LSTM

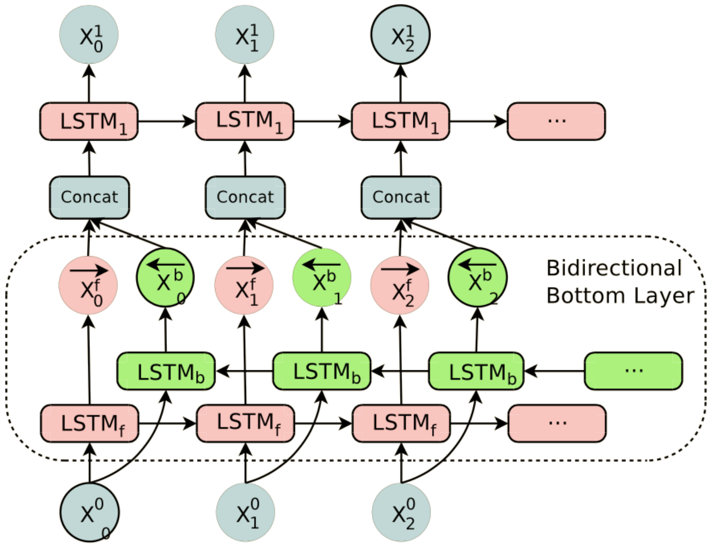
Useful

- Based on the element's past and future contexts.
- Proved to be especially useful when combined with LSTM RNN
- This is done by concatenating the outputs of two RNNs:
  - sequence from left to right
  - sequence from right to left

Mike Schuster and Kuldip K. Paliwal, "Bidirectional Recurrent Neural Networks", 1997

Alex Graves and Jurgen Schmidhuber, Framewise Phoneme "Classification with Bidirectional LSTM and Other Neural Network Architectures", 2005

Diagram



Equations

LSTM

Input gate → Incoming signal alter state of memory

$$i_t = \sigma(U_i x_t + W_i s_{t-1} + b_i)$$

Candidate Cell → States of memory at time t

$$\tilde{c_t} = tanh(U_c x_t + W_c s_{t-1} + b_c)$$

Forget Cell →activation of memory

$$f_t = \sigma\left(U_f x_t + W_f s_{t-1} + b_f\right)$$

State of cell → new state

$$c_t = f_t * c_{t-1} + i_t * \tilde{c_t}$$

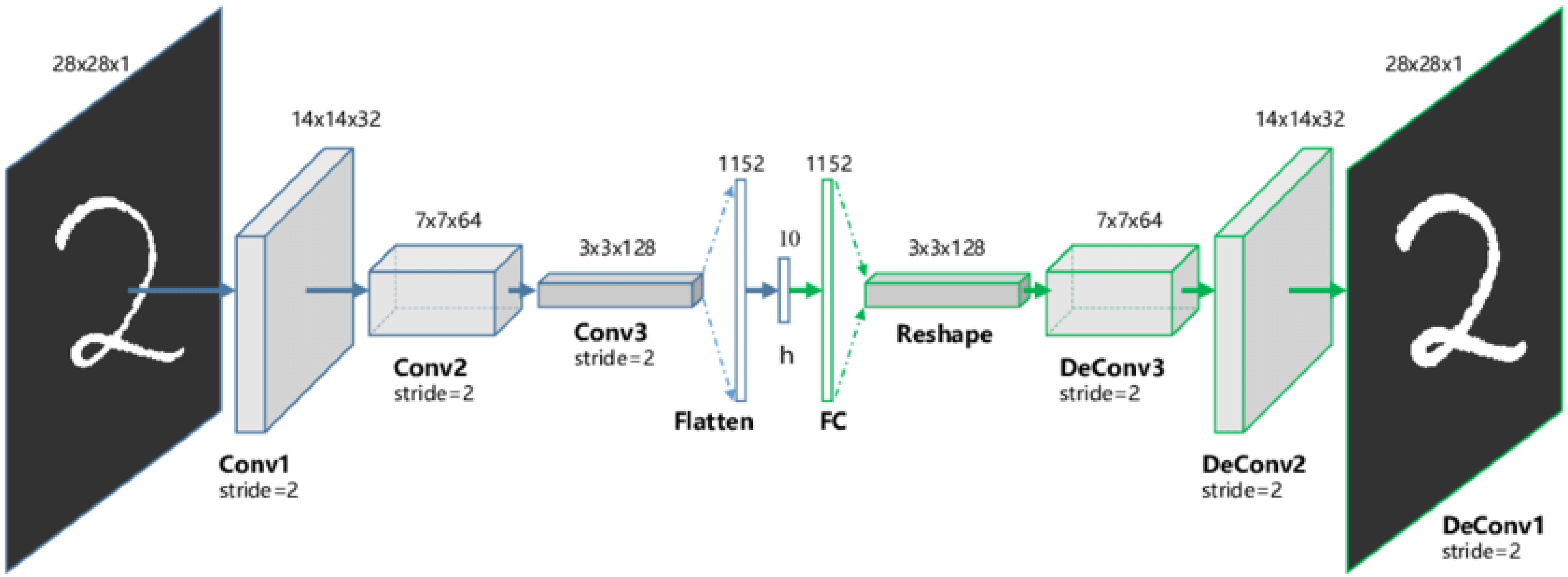
Output and hidden state

$$o_t = \sigma(U_o x_t + W_o s_{t-1} + V_o c_t + b_o)$$

$$s_t = o_t * tanh(c_t)$$

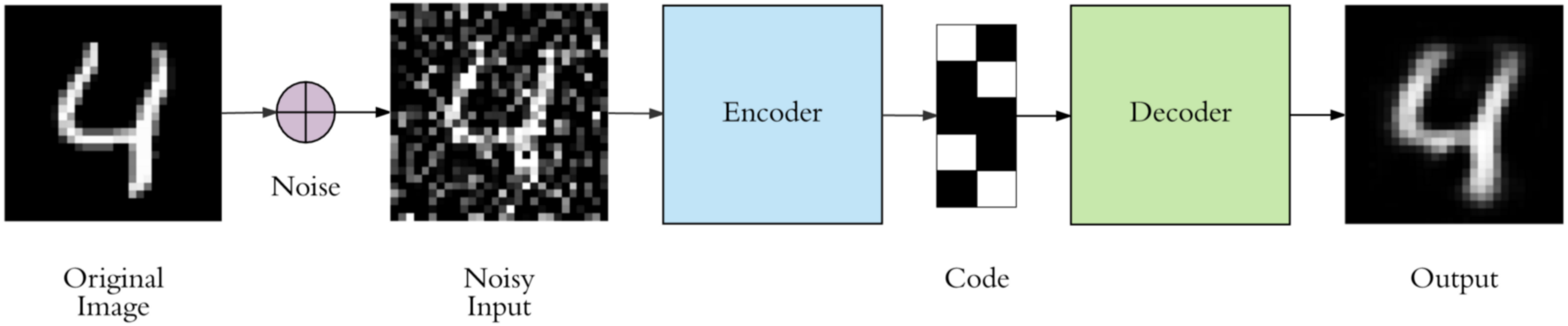
BRNN

```
# define the input
xi = K.Input((28,28))

# define the network
xn = K.layers.Dropout(rate)(xi)
xn = K.layers.Bidirectional(K.layers.LSTM( 32, return_sequences=True), merge_mode='ave')(xn)
xn = K.layers.BatchNormalization()(xn)
xn = K.layers.Bidirectional(K.layers.LSTM(32, return_sequences=False), merge_mode='ave')(xn)

# define the output
xo = K.layers.Dense(10, activation='softmax')(xn)

model = K.Model(inputs=[xi], outputs=[xo])
print(model.summary())
```



Autoencoder Neural Networks

Autoencoder Networks



At its most basic, an autoencoder is a neural network trained to copy its input to its output. Internally consists of two parts: an encoder function f(x) and a decoder function g(h). Instead of simply learning to copy, autoencoders are desgined to be unable to copy perfectly. Beacuse the model is forced to prioritize which aspects of the input should be copied, it often learns useful properties of the data.

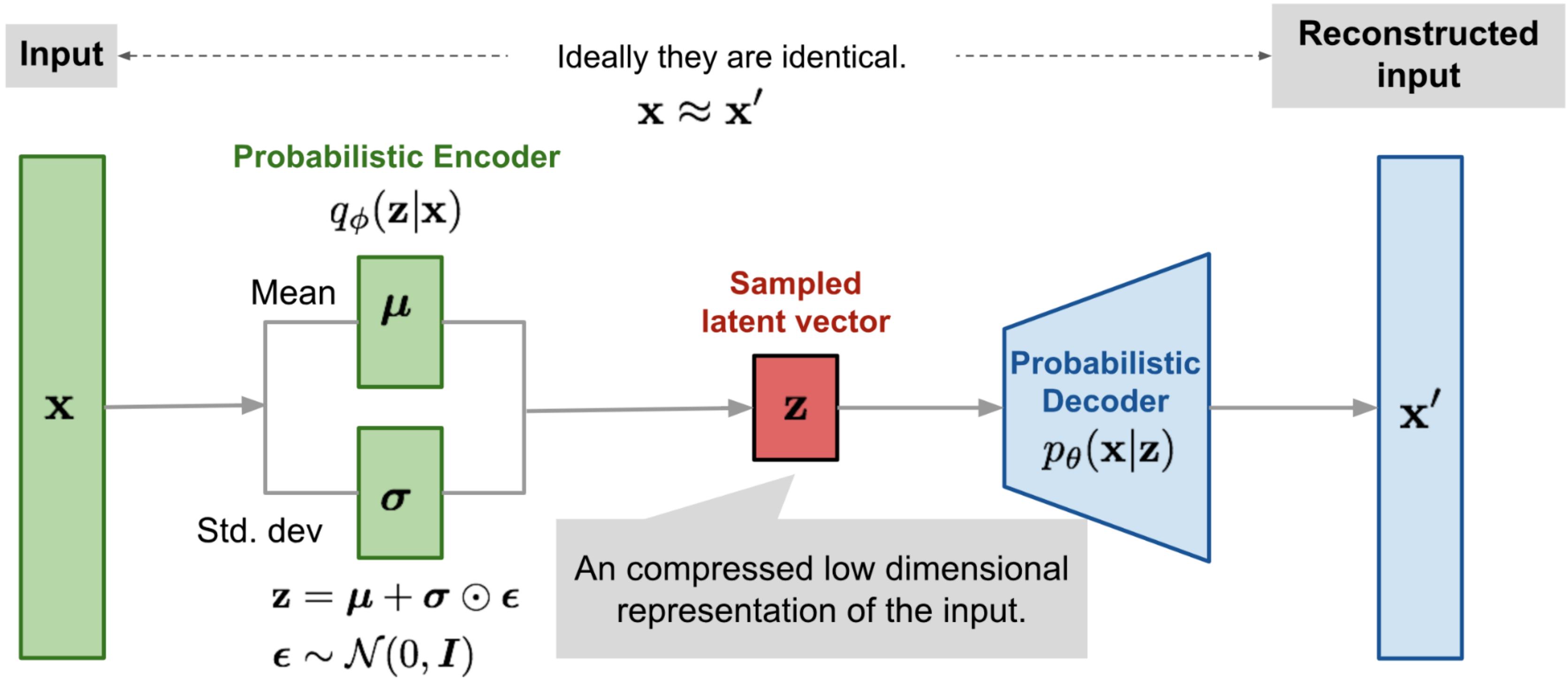
Autoencoder

```python
import tensorflow.keras.layers as L
import tensorflow.keras.backend as bk

def AutoEncoder(input_shape, layer_filters, latent_dim):
    # First, build the Encoder Model
    kernel_size = 3
    inputs = K.Input(shape=input_shape, name='encoder_input')
    x = inputs
    for filters in layer_filters:
        x = L.Conv2D( filters=filters,
                      kernel_size=kernel_size,
                      activation='relu',
                      padding='same')(x)

    shape = bk.int_shape(x)
    x = L.Flatten()(x)
    latent = L.Dense(latent_dim, name='latent_vector')(x)

    x = L.Dense(shape[1]*shape[2]*shape[3])(latent)
    x = L.Reshape((shape[1], shape[2], shape[3]))(x)
    for filters in layer_filters[::-1]:
        x = L.Conv2DTranspose( filters=filters,
                               kernel_size=kernel_size,
                               activation='relu',
                               strides=1,
                               padding='same')(x)
    x = L.Conv2D(filters=1,
                 kernel_size=kernel_size,
                 padding='same')(x)
    outputs = L.Activation('sigmoid', name='decoder_output')(x)
    decoder = K.Model(inputs, outputs, name='auto_encoder')
    return decoder
```

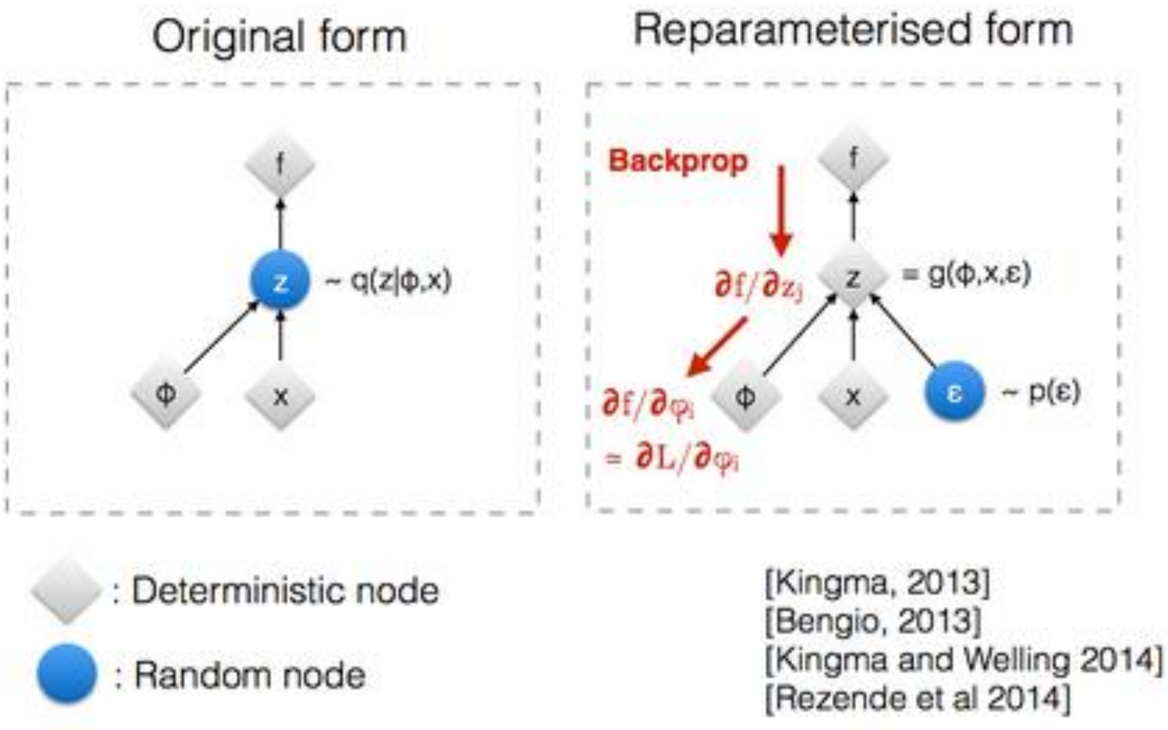
Autoencoder

```
# Autoencoder(input_shape(m,n), layer_filters, latent dim)
autoencoder = AutoEncoder((28,28,1),[32,16], 256)
autoencoder.summary()
autoencoder = AutoEncoder((100,50,1),[32,16], 256)
autoencoder.summary()


# Compile ,fit and evaluate model to the data
autoencoder.compile(loss='mse', optimizer='adam')
autoencoder.fit(x = train_images,
                y = train_labels,
                epochs=5,
                batch_size=32,
                shuffle=True,
                validation_data = (test_images, test_labels),
                verbose=2)
```

Variational Autoencoder



VAE

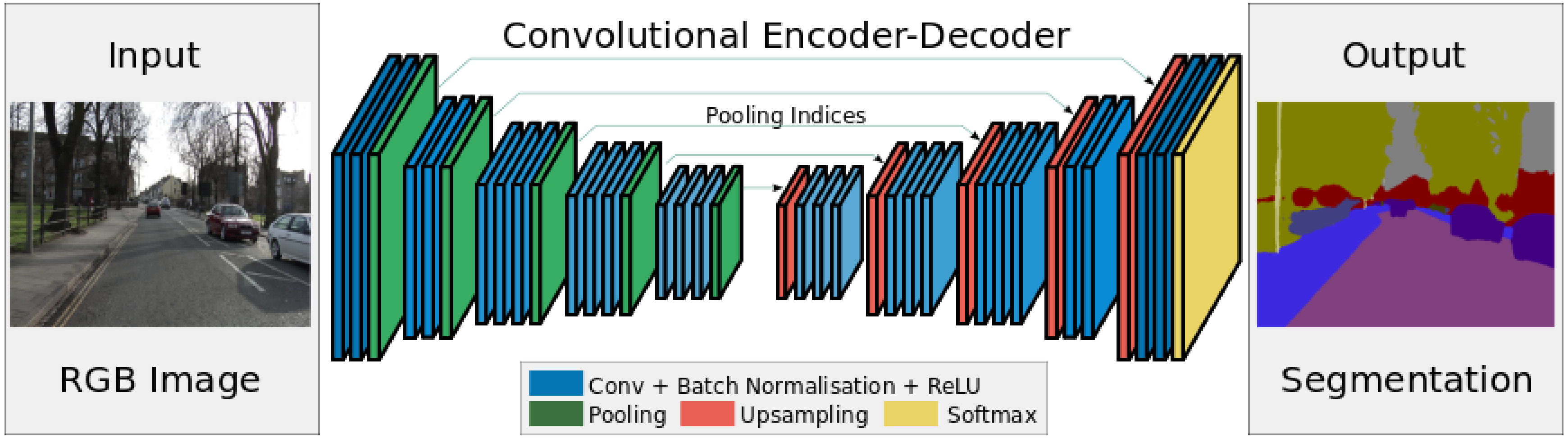



```python
def sampling(args):
    z_mean, z_log_var = args
    batch = bk.shape(z_mean)[0]
    dim = bk.int_shape(z_mean)[1]
    # by default, random_normal has mean = 0 and std = 1.0
    epsilon = bk.random_normal(shape=(batch, dim))
    return z_mean + bk.exp(0.5 * z_log_var) * epsilon
```

VAE

```
def VAE(input_shape, latent_dim):
    # VAE model = encoder + decoder
    # build encoder model
    intermediate_dim = 512
    xi = K.Input(shape=input_shape, name='encoder_input')
    x1 = L.Conv2D(filters=64,
                  kernel_size=3,
                  strides=1,
                  activation='relu',
                  padding='valid')(xi)
    x1 = L.MaxPooling2D()(x1)

    shape = bk.int_shape(x1)
    x1 = L.Flatten()(x1)
    x1 = L.Dense(intermediate_dim, activation='relu')(x1)

    z_mean = L.Dense(latent_dim, name='z_mean')(x1)
    z_log_var = L.Dense(latent_dim, name='z_log_var')(x1)
    z = L.Lambda(sampling, output_shape=(latent_dim,), name='z')([z_mean, z_log_var])

    x2 = L.Dense(intermediate_dim, activation='relu')(z)
    x2 = L.Dense(shape[1] * shape[2] * shape[3])(x2)
    x2 = L.Reshape((shape[1], shape[2], shape[3]))(x2)
    x2 = L.UpSampling2D()(x2)

    x2 = L.Conv2DTranspose(filters=64,
                           kernel_size=3,
                           strides=1,
                           activation='relu',
                           padding='valid')(x2)
    xo = L.Conv2DTranspose(filters=1,
                           kernel_size=3,
                           padding='same')(x2)

    vae = K.Model(xi, xo, name='vae_mlp')
    return vae
```

U-Net Autoencoder



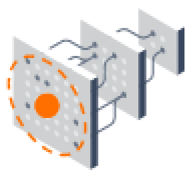
U-Net

```
def Autoencoder_Unet(input_shape, latent_dim):
    filters = [32,64,128,128,64,32]
    kernel_size = 3

    xi = K.Input(shape=input_shape, name='encoder_input')
    x1 = L.Conv2D(filters=filters[0],
                  kernel_size=(kernel_size,1),
                  activation='relu',
                  padding='valid')(xi)
    x1 = L.MaxPooling2D()(x1)
    x2 = L.Conv2D(filters=filters[1],
               kernel_size=kernel_size,
               activation='relu',
               padding='same')(x1)
    x3 = L.Conv2D(filters=filters[2],
               kernel_size=kernel_size,
               activation='relu',
               padding='same')(x2)
    shape = bk.int_shape(x3)
    xf = L.Flatten()(x3)
    xf = L.Dense(latent_dim, name='latent_vector')(xf)
```

U-Net

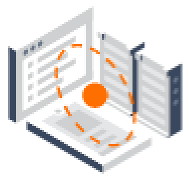
```
xf = L.Dense(shape[1] * shape[2] * shape[3])(xf)
xf = L.Reshape((shape[1], shape[2], shape[3]))(xf)

x4 = L.Conv2DTranspose(filters=filters[3],
                       kernel_size=kernel_size,
                       activation='relu',
                       padding='same')(xf)
x4 = L.Add()([x3,x4])
x5 = L.Conv2DTranspose(filters=filters[4],
                       kernel_size=kernel_size,
                       activation='relu',
                       padding='same')(x4)
x5 = L.Add()([x2,x5])
x6 = L.Conv2DTranspose(filters=filters[5],
                       kernel_size=kernel_size,
                       activation='relu',
                       padding='same')(x5)
x6 = L.Add()([x1,x6])
x6 = L.UpSampling2D()(x6)
x6 = L.ZeroPadding2D((1,0))(x6)
xo = L.Conv2DTranspose(filters=1,
                          kernel_size=kernel_size,
                          padding='same')(x6)
xo = L.Activation('sigmoid', name='decoder_output')(xo)
autoencoder = K.Model(xi, xo, name='autoencoder')
return autoencoder
```

BREAK



Easy model building

Build and train ML models easily using intuitive high-level APIs like Keras with eager execution, which makes for immediate model iteration and easy debugging.



Robust ML production anywhere

Easily train and deploy models in the cloud, on-prem, in the browser, or on-device no matter what language you use.

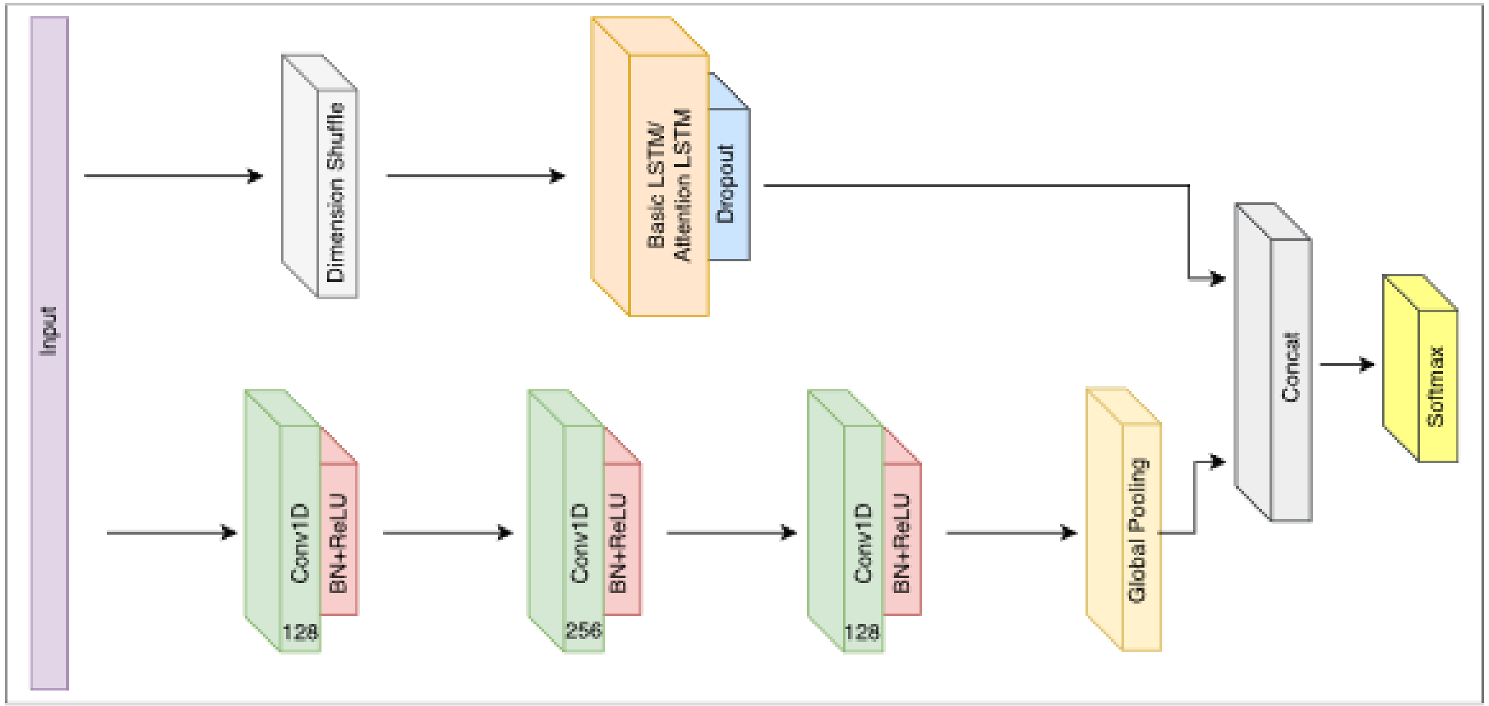
Powerful experimentation for research

A simple and flexible architecture to take new ideas from concept to code, to state-of-the-art models, and to publication faster.

Recent Works for Time Series*

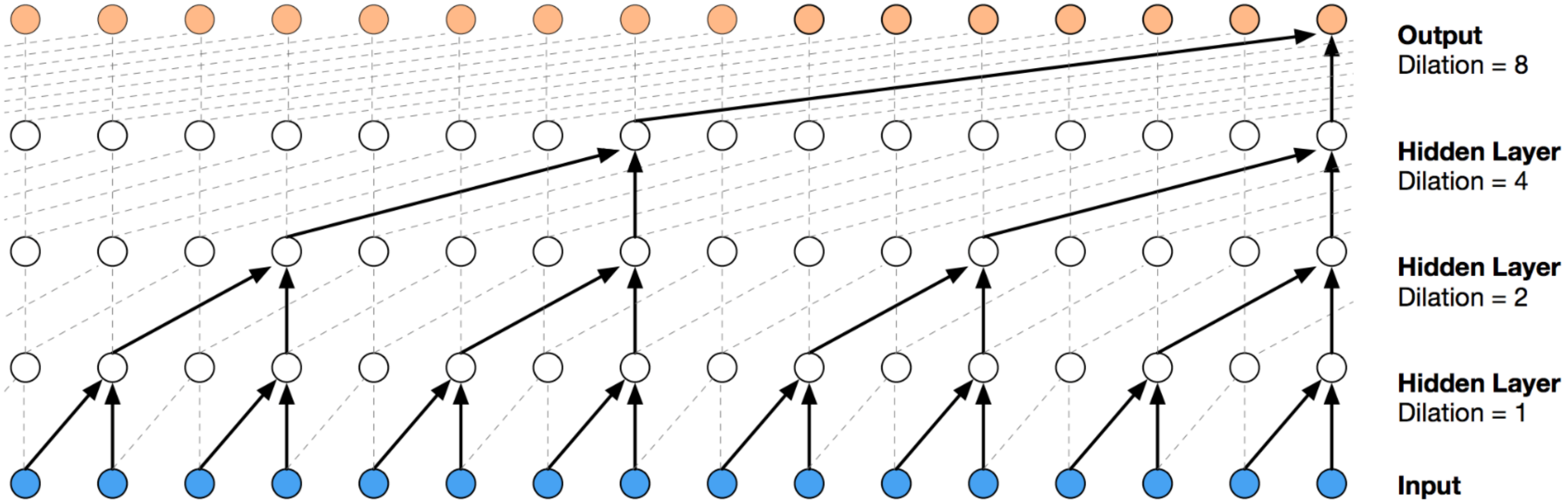
*available code

LSTM Fully Convolutional Networks for Time Series Classification



https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8141873
https://github.com/titu1994/LSTM-FCN

An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling



https://arxiv.org/pdf/1803.01271v2.pdf
https://github.com/philipperemy/keras-tcn

Thanks

Contact:

patoalejor@gmail.com