# BST 261: Data Science II
# Lecture 12
## Recurrent Neural Networks (RNNs) Continued

Heather Mattie
Harvard T.H. Chan School of Public Health
Spring 2 2020

# Recipe of the Day!

Butternut Squash Bisque

# Paper Presentations

# Sequence to Sequence Learning with Neural Networks

Ilya Sutskever, Oriol Vinyals & Quoc V. Le

## Introduction & Motivation

- Deep Neural Networks are flexible and powerful but can only be applied to problems with fixed dimensionality of inputs and targets.
- Many important problems are expressed with sequences whose lengths are unknown like speech recognition and machine translation.

Need: A domain-independent method for mapping sequences

- RNNs are difficult to train when there are long term dependencies in sequences.

Need: Models that can handle long range temporal dependencies

## Proposed Solution/Model

- **Problem:** Sequences are a challenge for DNNS because they have no fixed dimensionality and they encounter problems in long-term dependencies.

- **Solution:** Long Short-term Memory (LSTM) models

This model differs to models in the literature in three ways:

- Incorporate 2 LSTM models
  a) One LSTM reads input sequence => large fixed dimensional vector
  b) Another LSTM to extract the output sequence

- Deep LSTMs with 4 layers

- Reversing the order of the the input sentence into second LSTM

# The Model

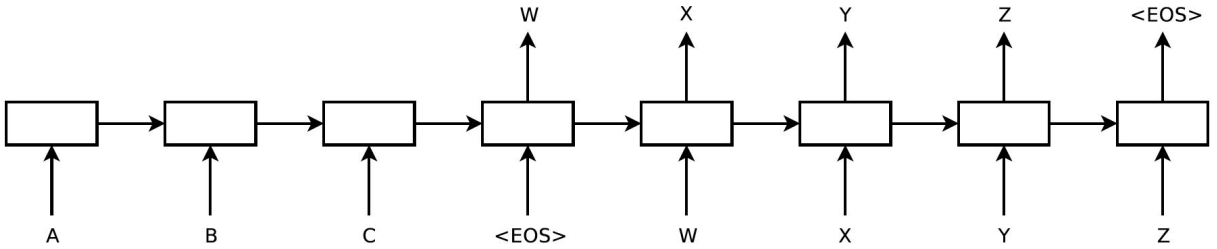



Figure 1: Our model reads an input sentence "ABC" and produces "WXYZ" as the output sentence. The model stops making predictions after outputting the end-of-sentence token. Note that the LSTM reads the input sentence in reverse, because doing so introduces many short term dependencies in the data that make the optimization problem much easier.

## Experiment

- **Dataset:** WMT'14 English to French, trained the models on a subset of 12M sentences consisting of 348M French words and 304M English words, fixed vocabulary for both languages 160,000 English of the most frequent words and 80,000 French
- **Training details:** 4 layers for each LSTM (with 1000 cells in each layer, output activation: naïve softmax, 380M parameters)

- LSTM work better on reversed input sentences

Introduction of many short term dependencies to the dataset (reduce "minimal time lag") and better memorization utilization

## Results

| Method | test BLEU score (ntst14) |
|---|---|
| Bahdanau et al. [2] | 28.45 |
| Baseline System [29] | 33.30 |
| Single forward LSTM, beam size 12 | 26.17 |
| Single reversed LSTM, beam size 12 | 30.59 |
| Ensemble of 5 reversed LSTMs, beam size 1 | 33.00 |
| Ensemble of 2 reversed LSTMs, beam size 12 | 33.27 |
| Ensemble of 5 reversed LSTMs, beam size 2 | 34.50 |
| Ensemble of 5 reversed LSTMs, beam size 12 | **34.81** |

Table 1: The performance of the LSTM on WMT'14 English to French test set (ntst14). Note that an ensemble of 5 LSTMs with a beam of size 2 is cheaper than of a single LSTM with a beam of size 12.

| Method | test BLEU score (ntst14) |
|---|---|
| Baseline System [29] | 33.30 |
| Cho et al. [5] | 34.54 |
| State of the art [9] | **37.0** |
| Rescoring the baseline 1000-best with a single forward LSTM | 35.61 |
| Rescoring the baseline 1000-best with a single reversed LSTM | 35.85 |
| Rescoring the baseline 1000-best with an ensemble of 5 reversed LSTMs | **36.5** |
| Oracle Rescoring of the Baseline 1000-best lists | ~45 |

Table 2: Methods that use neural networks together with an SMT system on the WMT'14 English to French test set (ntst14).

## Attractive Feature of the Model (1)

- **Performance on long sentences**

| Type | Sentence |
|---|---|
| **Our model** | Ulrich UNK , membre du conseil d' administration du constructeur automobile Audi , affirme qu' il s' agit d' une pratique courante depuis des années pour que les téléphones portables puissent être collectés avant les réunions du conseil d' administration afin qu' ils ne soient pas utilisés comme appareils d' écoute à distance . |
| **Truth** | Ulrich Hackenberg , membre du conseil d' administration du constructeur automobile Audi , déclare que la collecte des téléphones portables avant les réunions du conseil , afin qu' ils ne puissent pas être utilisés comme appareils d' écoute à distance , est une pratique courante depuis des années . |

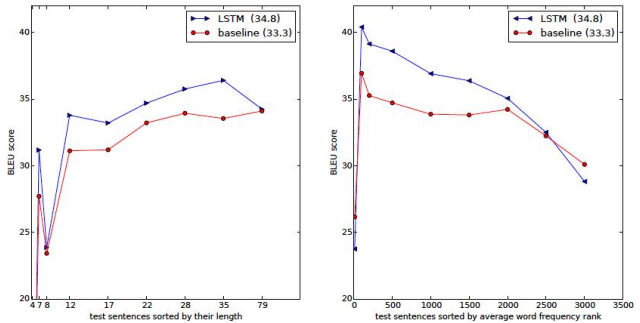



Figure 3: The left plot shows the performance of our system as a function of sentence length, where the x-axis corresponds to the test sentences sorted by their length and is marked by the actual sequence lengths. There is no degradation on sentences with less than 35 words, there is only a minor degradation on the longest sentences. The right plot shows the LSTM's performance on sentences with progressively more rare words, where the x-axis corresponds to the test sentences sorted by their "average word frequency rank".

## Attractive Feature of the Model (2)

- **Sequence to a fixed vector**

Learned representations are sensitive to word order but fairly insensitive to active/passive voices


Figure 2: The figure shows a 2-dimensional PCA projection of the LSTM hidden states that are obtained after processing the phrases in the figures. The phrases are clustered by meaning, which in these examples is primarily a function of word order, which would be difficult to capture with a bag-of-words model. Notice that both clusters have similar internal structure.

## Concluding Points

Improving sequence to sequence learning and contributing to scientific knowledge by:

- Using an all DNN model and no SMT paradigm
- Vector representations of sentences are sensitive to word order, sensible phrases
- Invariant to active/passive voices
- Data transformation (reversing input sentence) improves performance
- Overcome problems with translation of long sentences

This simple, straightforward, relatively unoptimized approach outperforms mature phased-based SMT and further work could improve machine translation.

# Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network

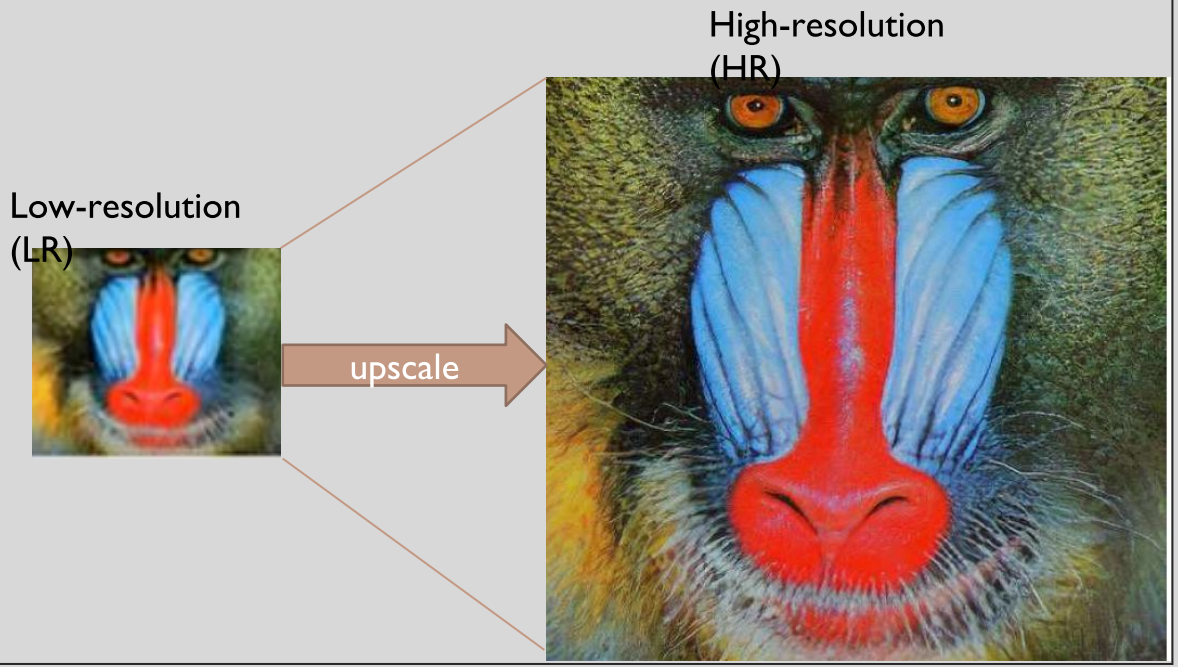
Christian Ledig, Lucas Theis, Ferenc Huszar, Jose Caballero, Andrew Cunningham, Alejandro Acosta, Andrew Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, Wenzhe Shi

BST 261 Paper Presentation by: Zifan Wang 4/29/2020

Background: Image Super-Resolution

**Super-resolution (SR):** Estimating a high-resolution (HR) image from its low-resolution (LR) counterpart.

Applications:
- Satellite imaging
- Media content
- Medical imaging
- Surveillance
- …



## Main problem of SR for high upscaling factors

- **Texture detail** in the reconstructed SR images is typically **absent**.
- **Commonly used** optimization target of supervised SR algorithms: **minimizing the mean squared error(MSE)** between the recovered HR image and the ground truth.

  - Convenient: minimizing the MSE also maximizes the peak signal-to-noise ratios (PSNR)

  but…

  - The ability of MSE (and PSNR) to capture **perceptually relevant differences** (e.g. high texture detail) is **very limited**, as they are defined based on pixel-wise image differences

PSNR / SSIM

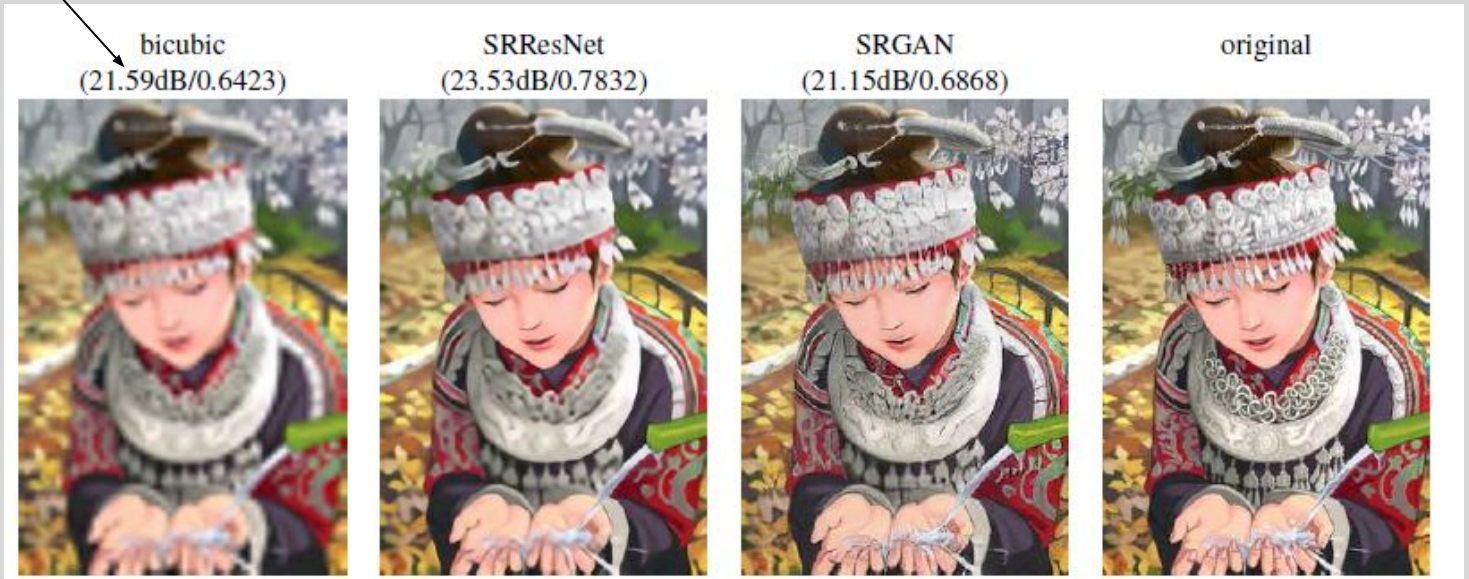



Figure 2: From left to right: bicubic interpolation, deep residual network optimized for MSE, deep residual generative adversarial network optimized for a loss more sensitive to human perception, original HR image. Corresponding PSNR and SSIM are shown in brackets. [4× upscaling]

Perceptual difference between a super-resolved and original image  the recovered image is **not photo-realistic**

Key contribution of this paper

- 1. **SRResNet**:

  Set a new state of the art for image SR with high upscaling factors (4x) as measured by peak signal-to-noise ratio (PSNR) and structural similarity (SSIM) with 16 blocks deep ResNet (SRResNet) optimized for MSE.

- 2. **SRGAN**

  A GAN-based network optimized for a new perceptual loss.

  Replaced the MES-based content loss with a loss calculated on feature maps of the VGG network, combined with a discriminator.

- 3. Results

  Confirmed with an extensive mean opinion score (MOS) test on images from three public benchmark datasets that SRGAN is the new state of the art for the estimation of photo-realistic SR images with high upscaling factors (4x).

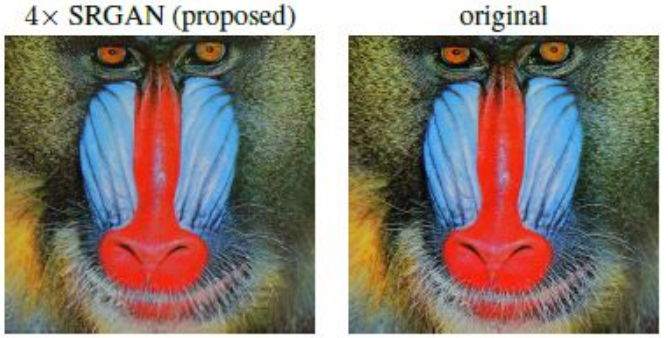



Figure 1: Super-resolved image (left) is almost indistinguishable from original (right). [4× upscaling]

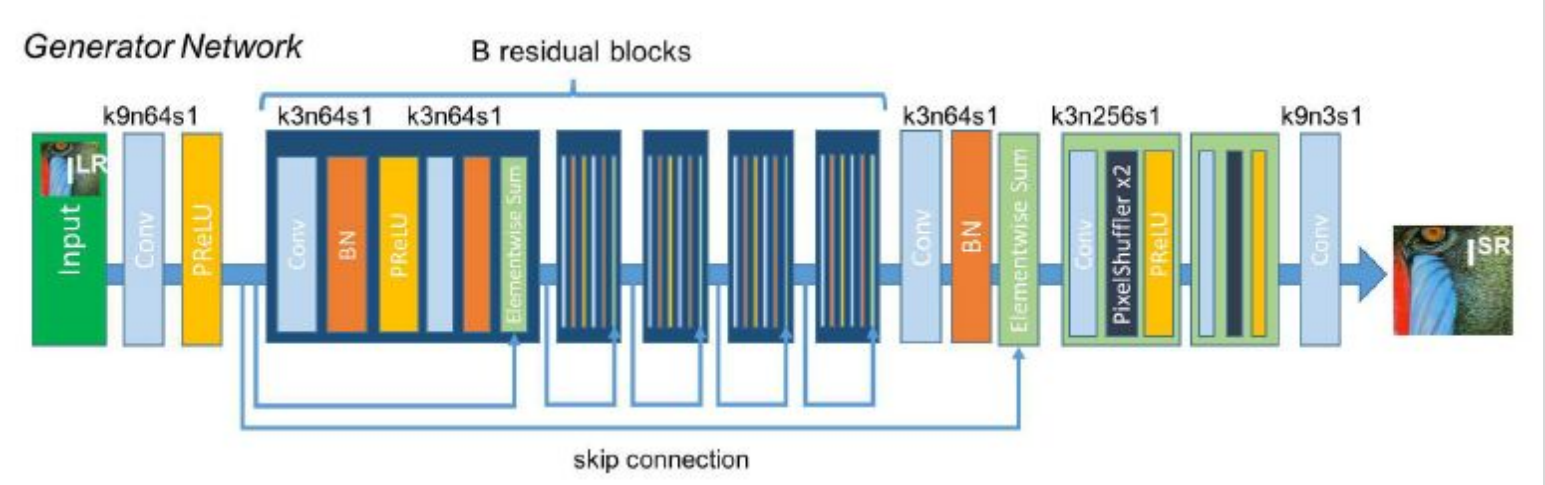
SRResNet

- Deep residual CNN for image SR
  - Trained on ImageNet (350k images)
  - Optimized for MSE

The pixel-wise **MSE loss** is calculated as:

$$l_{MSE}^{SR} = \frac{1}{r^2WH} \sum_{x=1}^{rW} \sum_{y=1}^{rH} (I_{x,y}^{HR} - G_{\theta_G}(I^{LR})_{x,y})^2 \qquad (4)$$

Architecture




- Upscaling using sub-pixel convolution

From MSE to Perceptual Loss functions

- **The problem of minimizing MSE**: encourages finding pixel-wise averages of plausible solutions which are typically **overly-smooth** and thus have **poor perceptual quality**.

Tackling this problem by…

- **Generative adversarial networks (GANs):**

- A powerful framework for generating plausible-looking natural images with high perceptual quality
- Use of features extracted from a pretrained VGG network instead of low-level pixel-wise error measures
- Very deep ResNet architecture using the concept of GANs to form a perceptual loss function for photo-realistic SISR

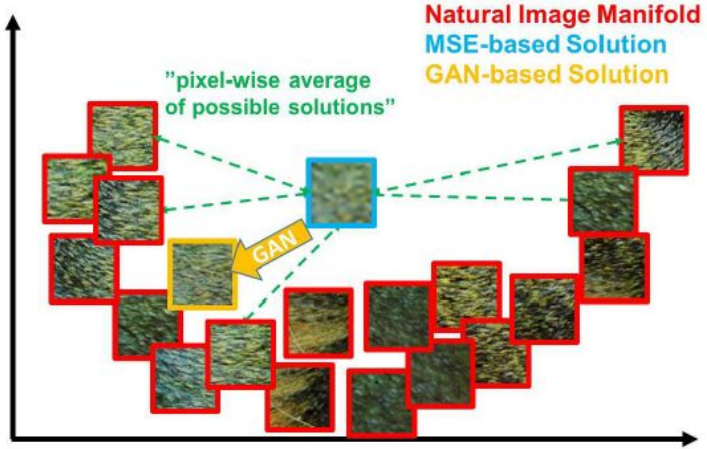



Figure 3: Illustration of patches from the natural image manifold (red) and super-resolved patches obtained with MSE (blue) and GAN (orange). The MSE-based solution appears overly smooth due to the pixel-wise average of possible solutions in the pixel space, while GAN drives the reconstruction towards the natural image manifold producing perceptually more convincing solutions.

SRGAN

- MSE:

$$l^{SR} = l^{SR}_{MSE}$$

Content loss: ensures **pixel-level** content preserved

- Perceptual loss:

Content loss: ensures **high-level** content preserved

$$l^{SR} = \underbrace{\underbrace{l^{SR}_{X}}_{\text{content loss}} + \underbrace{10^{-3} l^{SR}_{Gen}}_{\text{adversarial loss}}}_{\text{perceptual loss (for VGG based content losses)}}$$

Adversarial loss: ensures reconstructed images **look real**

$$l^{SR}_{VGG/i.j} = \frac{1}{W_{i,j} H_{i,j}} \sum_{x=1}^{W_{i,j}} \sum_{y=1}^{H_{i,j}} (\phi_{i,j}(I^{HR})_{x,y} - \phi_{i,j}(G_{\theta_G}(I^{LR}))_{x,y})^2$$

Architecture

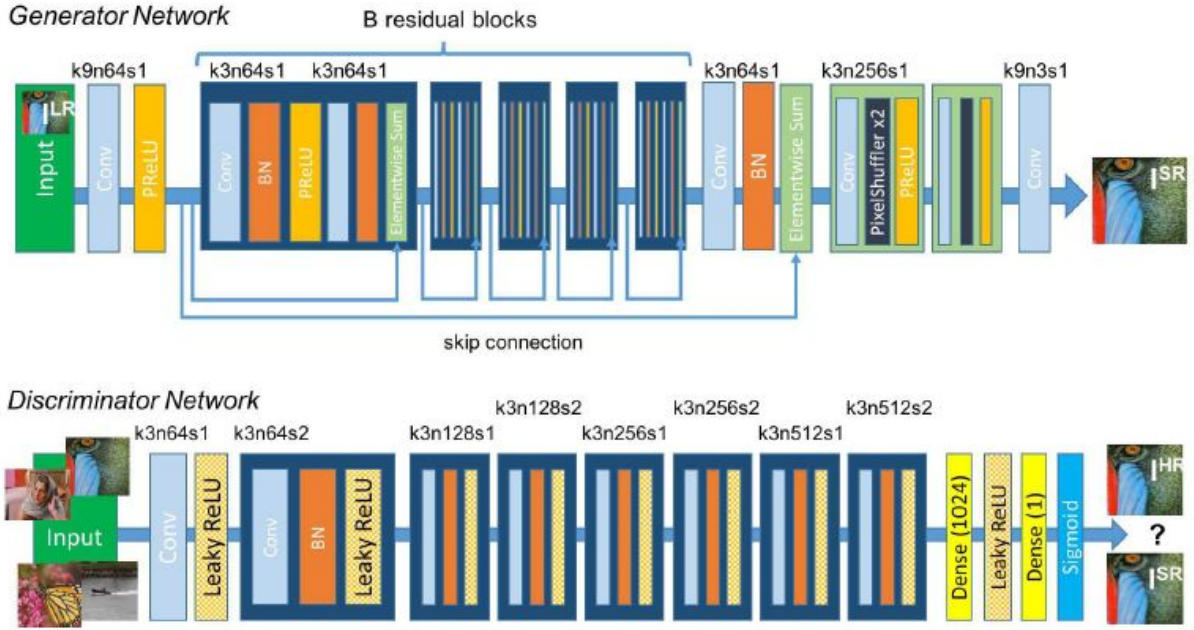



Figure 4: Architecture of Generator and Discriminator Network with corresponding kernel size (k), number of feature maps (n) and stride (s) indicated for each convolutional layer.

## Experiments & Results

- Performed experiments on three widely used bench-mark datasets: Set5, Set14, and BSD100
- All experiments performed with a scale factor of 4x between LR and HR images
- **PSNR and SSIM compared across**: SRResNet (MSE vs. VGG) and SRGAN variants
- **Mean opinion score (MOS) testing**: to quantify the ability of different approaches to reconstruct perceptually convincing images.

  26 human raters:

  - give scores 1(bad) to 5 (excellent) to the SR images quality
  - each rater rated > 1000 images

Table 1: Performance of different loss functions for SRResNet and the adversarial networks on Set5 and Set14 benchmark data. MOS score significantly higher ($p < 0.05$) than with other losses in that category*. [4× upscaling]

| | SRResNet- | | SRGAN- | | |
|---|---|---|---|---|---|
| **Set5** | MSE | VGG22 | MSE | VGG22 | VGG54 |
| PSNR | 32.05 | 30.51 | 30.64 | 29.84 | 29.40 |
| SSIM | 0.9019 | 0.8803 | 0.8701 | 0.8468 | 0.8472 |
| MOS | 3.37 | 3.46 | 3.77 | 3.78 | 3.58 |
| **Set14** | | | | | |
| PSNR | 28.49 | 27.19 | 26.92 | 26.44 | 26.02 |
| SSIM | 0.8184 | 0.7807 | 0.7611 | 0.7518 | 0.7397 |
| MOS | 2.98 | 3.15* | 3.43 | 3.57 | 3.72* |

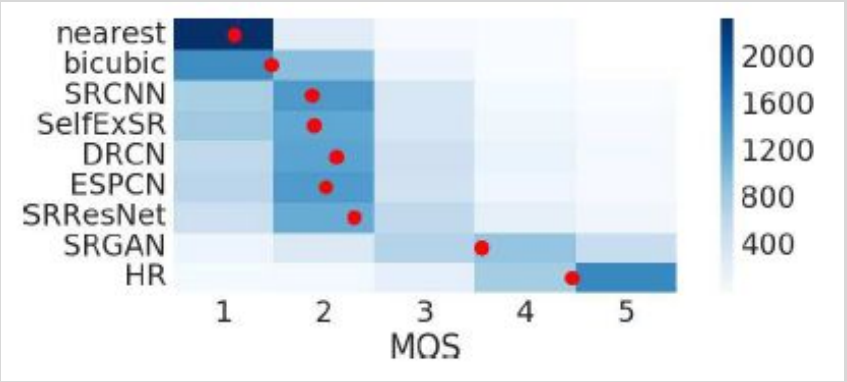



Figure 5: Color-coded distribution of MOS scores on BSD100. For each method 2600 samples (100 images × 26 raters) were assessed. Mean shown as red marker, where the bins are centered around value i. [4× upscaling]

SRResNet-MSE SRGAN-VGG54

Table 2: Comparison of NN, bicubic, SRCNN [8], SelfExSR [30], DRCN [33], ESPCN [47], **SRResNet**, **SRGAN**-VGG54 and the original HR on benchmark data. Highest measures (PSNR [dB], SSIM, MOS) in bold. [4× upscaling]

| **Set5** | nearest | bicubic | SRCNN | SelfExSR | DRCN | ESPCN | **SRResNet** | **SRGAN** | HR |
|---|---|---|---|---|---|---|---|---|---|
| PSNR | 26.26 | 28.43 | 30.07 | 30.33 | 31.52 | 30.76 | **32.05** | 29.40 | ∞ |
| SSIM | 0.7552 | 0.8211 | 0.8627 | 0.872 | 0.8938 | 0.8784 | **0.9019** | 0.8472 | 1 |
| MOS | 1.28 | 1.97 | 2.57 | 2.65 | 3.26 | 2.89 | 3.37 | **3.58** | 4.32 |
| **Set14** | | | | | | | | | |
| PSNR | 24.64 | 25.99 | 27.18 | 27.45 | 28.02 | 27.66 | **28.49** | 26.02 | ∞ |
| SSIM | 0.7100 | 0.7486 | 0.7861 | 0.7972 | 0.8074 | 0.8004 | **0.8184** | 0.7397 | 1 |
| MOS | 1.20 | 1.80 | 2.26 | 2.34 | 2.84 | 2.52 | 2.98 | **3.72** | 4.32 |
| **BSD100** | | | | | | | | | |
| PSNR | 25.02 | 25.94 | 26.68 | 26.83 | 27.21 | 27.02 | **27.58** | 25.16 | ∞ |
| SSIM | 0.6606 | 0.6935 | 0.7291 | 0.7387 | 0.7493 | 0.7442 | **0.7620** | 0.6688 | 1 |
| MOS | 1.11 | 1.47 | 1.87 | 1.89 | 2.12 | 2.01 | 2.29 | **3.56** | 4.46 |

Discussion

- **The choice of content loss** is of particular importance when aiming for photo-realistic solutions to the SR problem:
- Also, the ideal loss function **depends on the application**

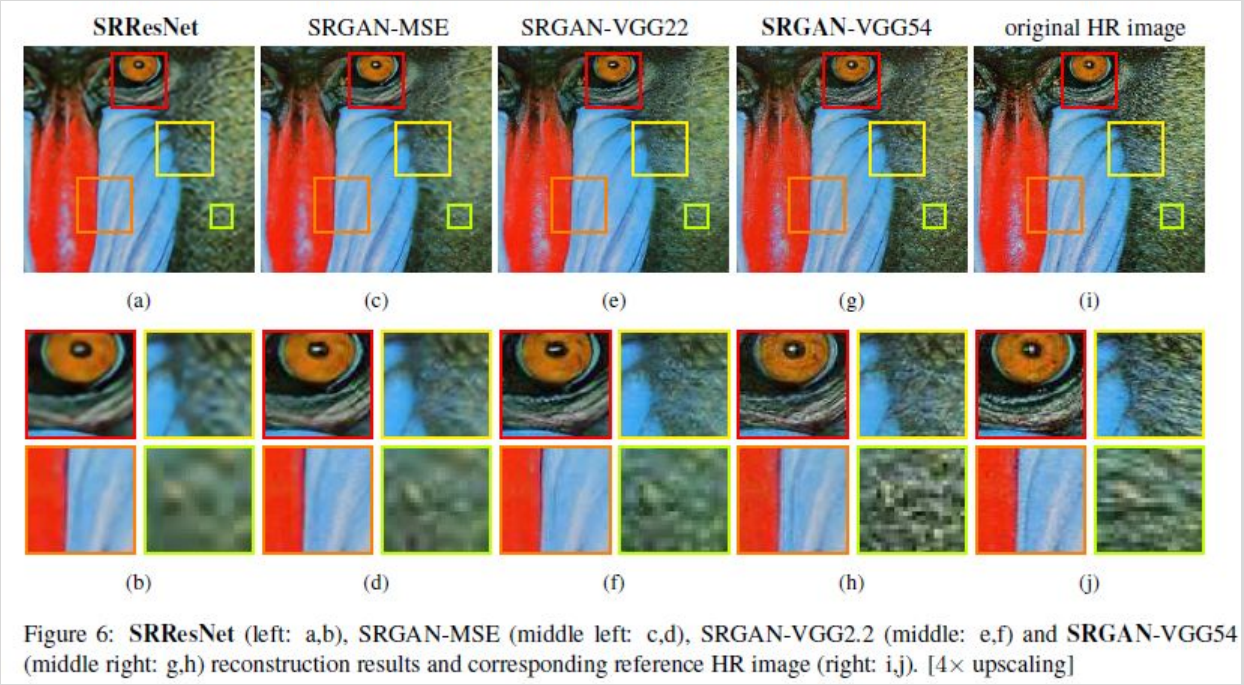



Figure 6: **SRResNet** (left: a,b), SRGAN-MSE (middle left: c,d), SRGAN-VGG2.2 (middle: e,f) and **SRGAN**-VGG54 (middle right: g,h) reconstruction results and corresponding reference HR image (right: i,j). [4× upscaling]

## Conclusion

- **SRResNet** as a deep residual network that sets a new state of the art on public benchmark datasets when evaluated with widely used **PSNR measure**.
- Highlighted some limitations of PSNR-focused image SR and introduced **SRGAN**, which augments the content loss function with an adversarial loss by training a GAN.
- Using extensive **MOS testing**, the authors confirmed that **SRGAN** reconstructions for large upscaling factors (4x) are **more photo-realistic** than reference methods.

Reference: Ledig, Christian, et al. "Photo-realistic single image super-resolution using a generative adversarial network." Proceedings of the IEEE conference on computer vision and pattern recognition. 2017.

# Deep Learning Predicts Tuberculosis Drug Resistance Status from Whole-Genome Sequencing Data

Michael L. Chen, Akshith Doddi, Jimmy Royer, Luca Freschi, Marco Schito, Matthew Ezewudo, Isaac Kohane, Andrew Beam, and Maha Farhat

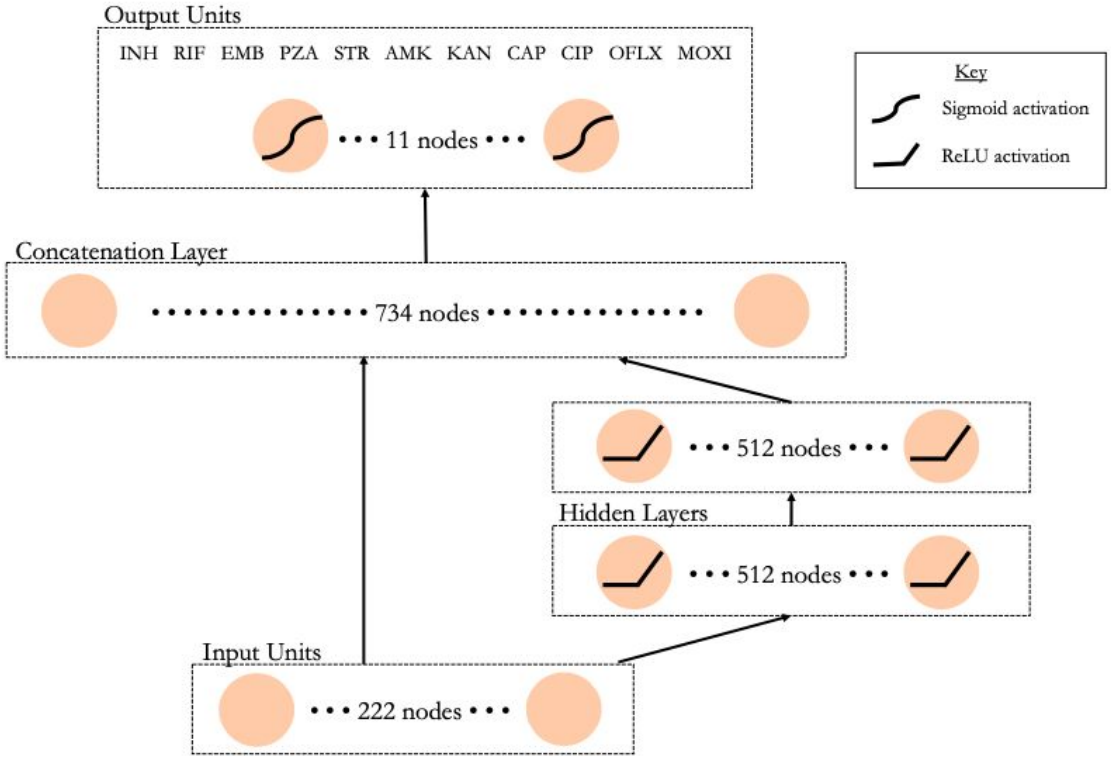
Presented by: Sarah Bolongaita

# Introduction

- TB is among the **top 10 causes of mortality** worldwide and the leading cause of death from a single infectious agent, *Mycobacterium tuberculosis* (MTB)
- Growing use of antibiotics has led to increased prevalence of **drug resistance**
- Diagnosing drug resistance is a **barrier to appropriate TB treatment**
    - Culture-based methods - Resource intensive, time consuming
    - Targeted molecular diagnostics (e.g. GeneXpert) - Limited sensitivity, smaller scope
    - Whole genome sequencing - Larger scope, wide range of sensitivity

# Wide and Deep Neural Network (WDNN)

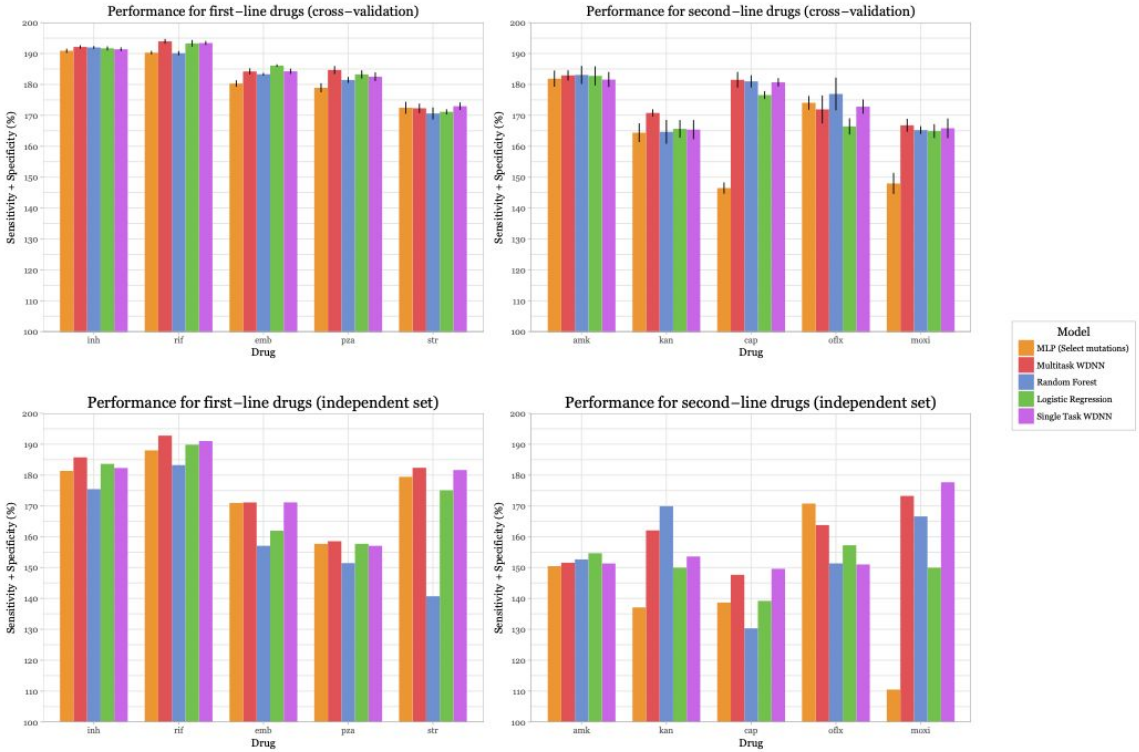
- **Wide and deep structure** allows the inclusion of prior information about the genetic etiology of drug resistance
  - **Wide portion (logistic regression)** - Allows the effect of individual mutations to be easily learned
  - **Deep portion (MLP)** - Allows for arbitrarily complex epistatic effects (i.e. modifier genes) to influence predictions
- **Multitask structure** allows drugs which have less phenotypic data to borrow information about resistance pathways form drugs that have more phenotyped isolates



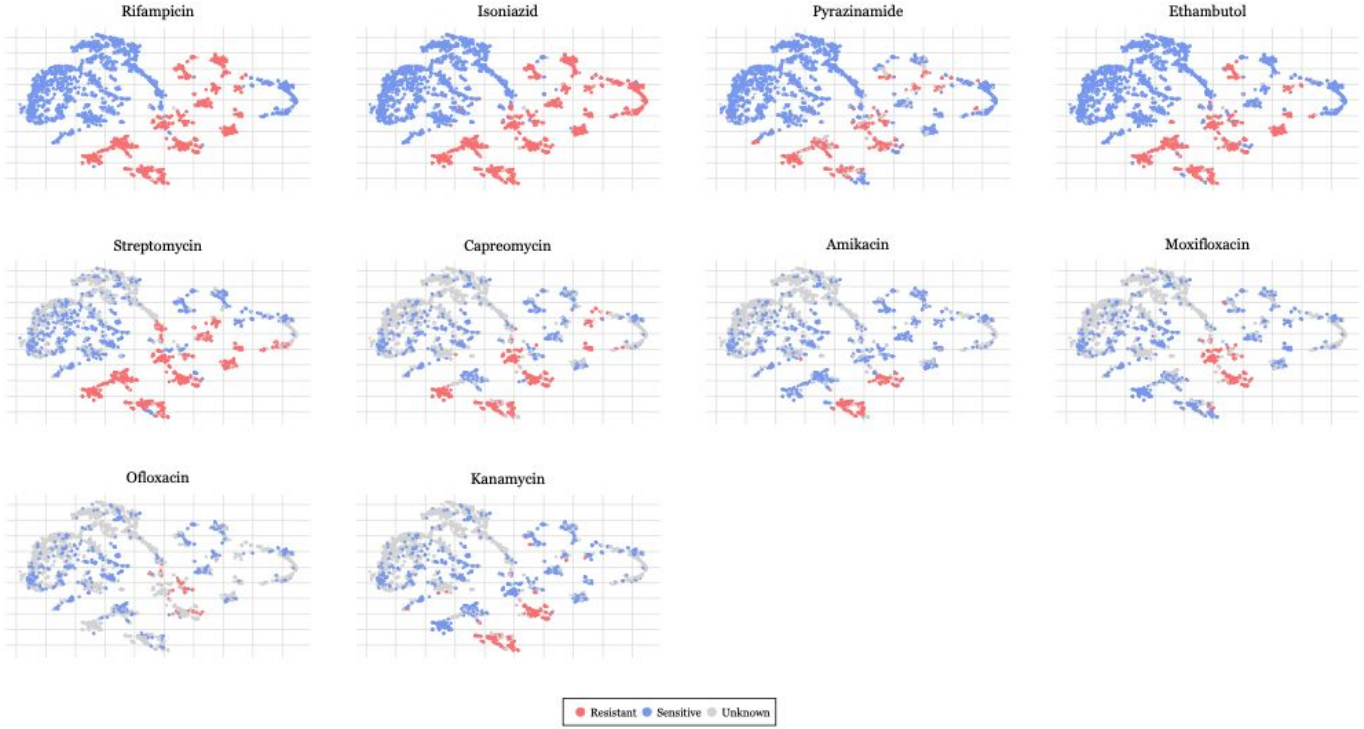

# Wide and Deep Neural Network (WDNN)

# Data and Methods

- Pooled data from the WHO network of supranational reference laboratories and the Relational Sequencing TB Knowledgebase (ReSeqTB)
  - Training set - 3,601 MTB isolates
  - Validation set - 792 MTB isolates
- Use a **multitask wide and deep neural network (WDNN)** as a tool to predict phenotypic drug resistance for 10 TB drugs simultaneously
- Compared multitask WDNN performance with
  - Single task WDNN
  - Single task multilayer perceptron (MLP) trained on pre-selected mutations known to be resistance-determining for each drug
  - Baseline machine learning models: Random forest and regularized logistic regression


Performance for first−line drugs (cross−validation)
Performance for second−line drugs (cross−validation)
Performance for first−line drugs (independent set)
Performance for second−line drugs (independent set)
Sensitivity + Specificity (%)
Drug
inh
rif
emb
pza
str
amk
kan
cap
oflx
moxi
Model
MLP (Select mutations)
Multitask WDNN
Random Forest
Logistic Regression
Single Task WDNN

# Results


Rifampicin
Isoniazid
Pyrazinamide
Ethambutol
Streptomycin
Capreomycin
Amikacin
Moxifloxacin
Ofloxacin
Kanamycin
Resistant
Sensitive
Unknown

# Results

# Take Home

- Paper presented a new deep learning architecture - a multitask wide and deep neural network (WDNN) - to identify MTB resistance to 10 TB drugs
- Model out-performed other models for at least 7 out of 10 TB drugs demonstrating the efficacy of deep learning as a diagnostic tool for MTB resistance

# Long Short Term Memory (LSTM)

# Problems with RNNs

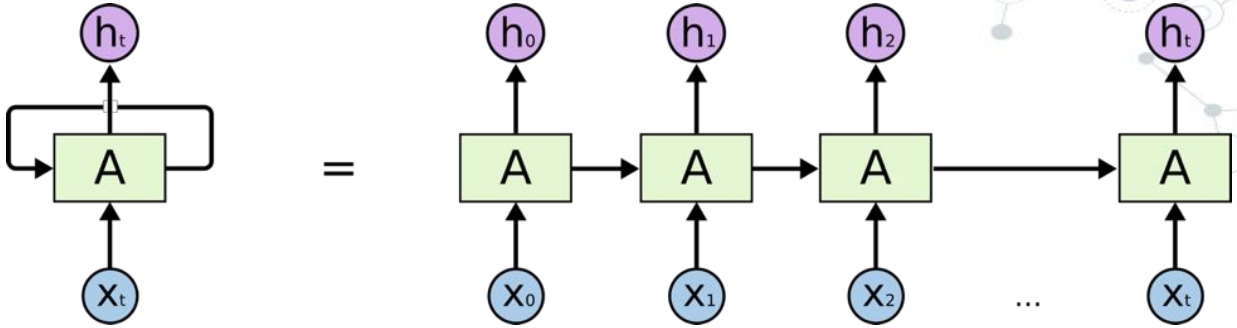
- Recall the formula for a generic RNN:

$$h_t = f(X_t W + h_{t-1} U + b)$$

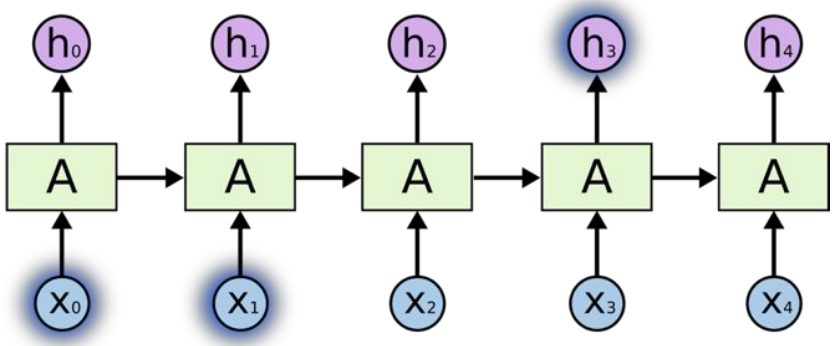
- What happens for really long sequences during backprop?
  - You multiply by the matrix U repeatedly
  - Largest eigenvalue > 1, gradient $\longrightarrow$ $\infty$ (explodes)
  - Largest eigenvalue < 1, gradient $\longrightarrow$ 0 (vanishes)

- This is known as the **vanishing or exploding gradient problem**

An “unrolled” RNN



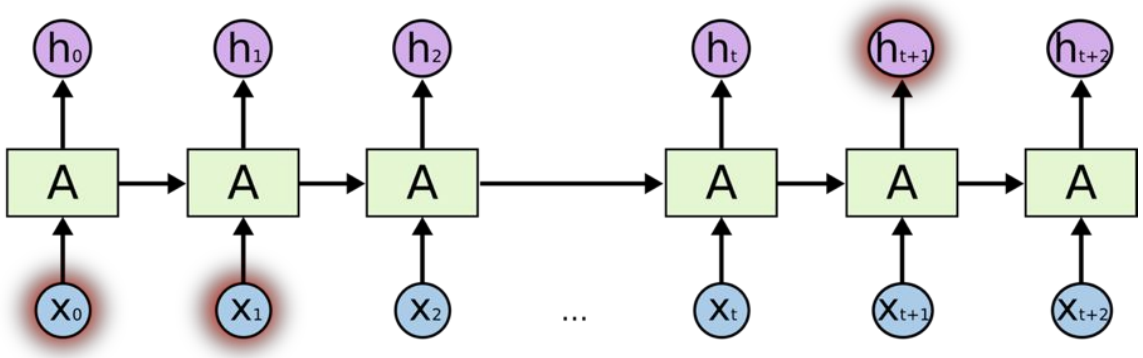

RNN where the output $h_3$ only depends on the input from $X_0$ and $X_1$
(The relevant information needed at $h_3$ comes from $X_0$ and $X_1$)
The gap between relevant information and the place it is needed is small



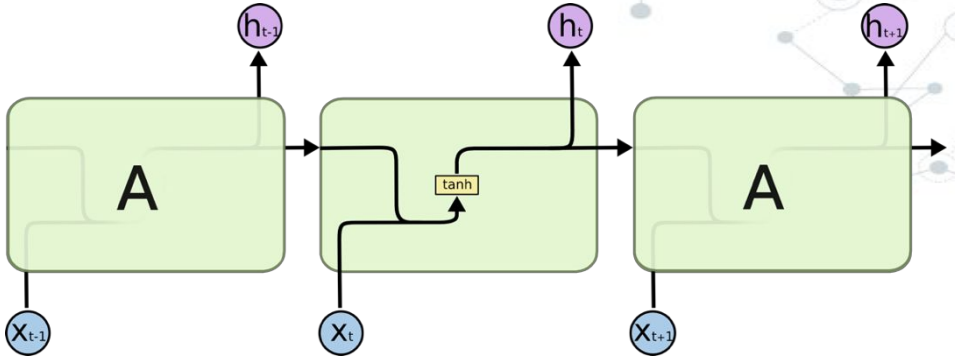

http://colah.github.io/posts/2015-08-Understanding-LSTMs/

RNN where the output $h_{t+1}$ is dependent on data inputs $X_0$ and $X_1$ that are too far for the gradient to carry

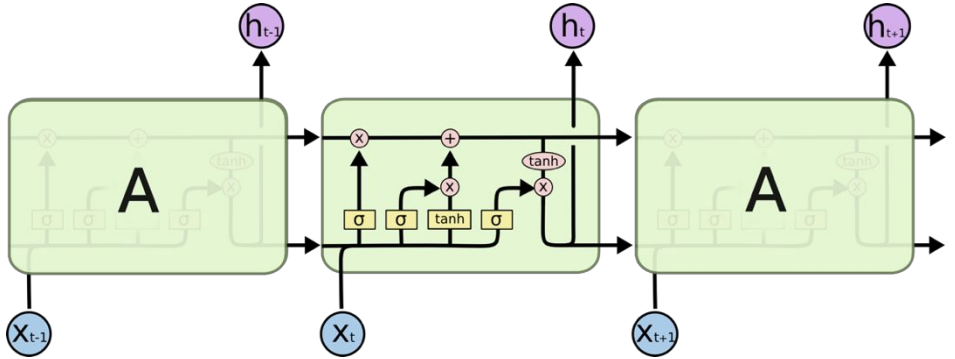
This is an example of a **long-term dependency** - RNNs struggle to learn to make connections when there are large gaps between the relevant information and where it is needed

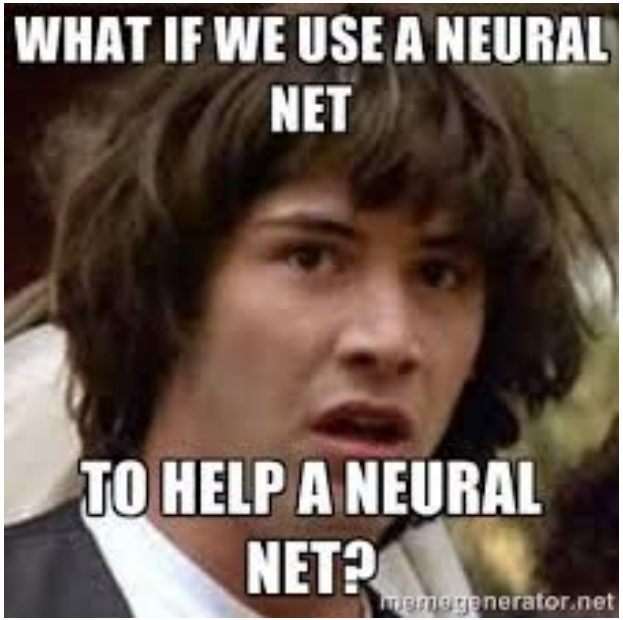


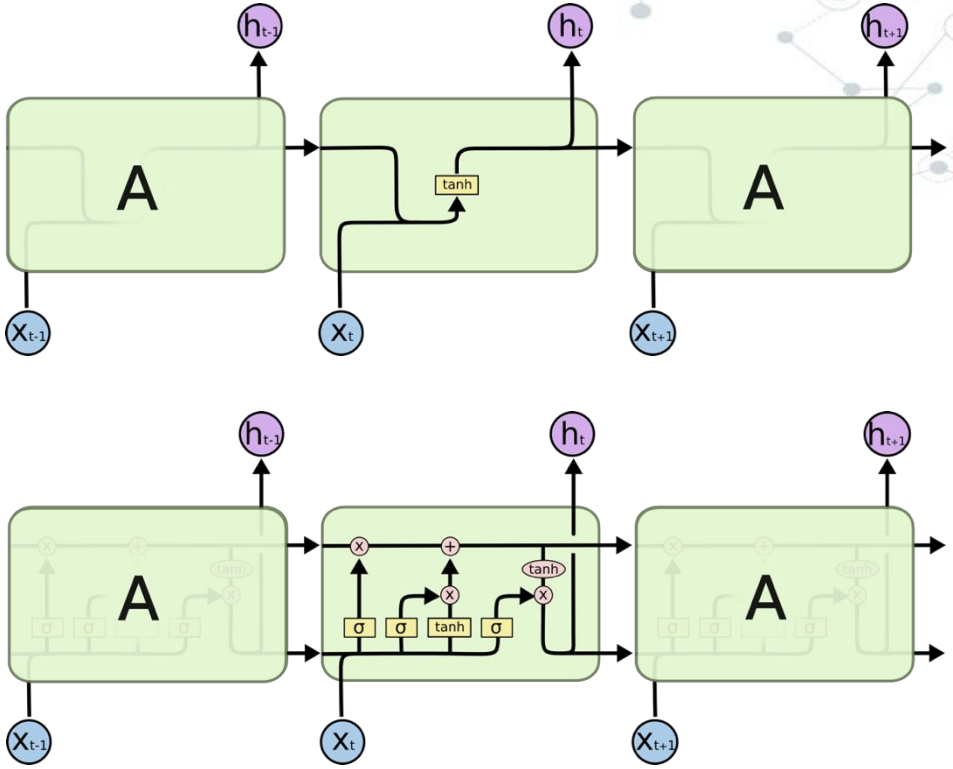
Simple, “vanilla” RNN:



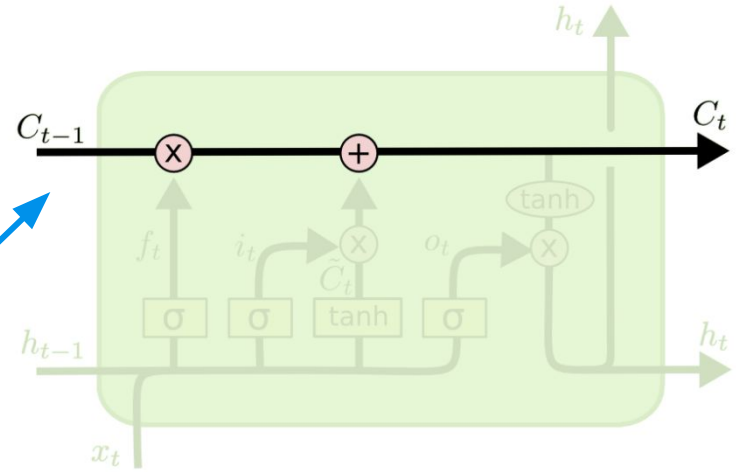

RNN with LSTM units:

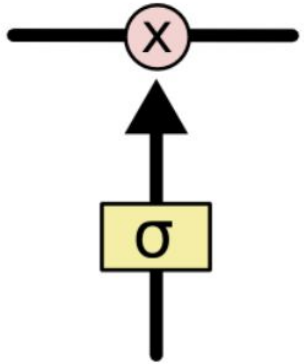



WHAT IF WE USE A NEURAL NET
TO HELP A NEURAL NET?
memegenerator.net


$h_{t-1}$
$h_t$
$h_{t+1}$
A
tanh
A
$x_{t-1}$
$x_t$
$x_{t+1}$
$h_{t-1}$
$h_t$
$h_{t+1}$
A
σ
σ
tanh
σ
tanh
A
$x_{t-1}$
$x_t$
$x_{t+1}$
Neural Network Layer
Pointwise Operation
Vector Transfer
Concatenate
Copy

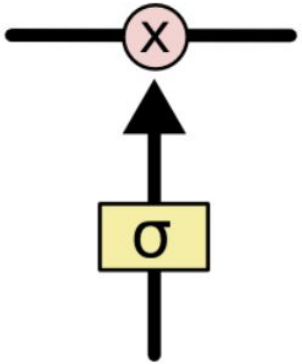
- LSTMs were explicitly designed to avoid the long-term dependency problem
- The key to LSTMs is the ability to let certain information through and carry it until it is deemed no longer useful (which may not happen)
- Information is carried through the sequence in the **cell state**, which acts as a conveyor belt or highway of information (memory of the network)
- Information is kept or forgotten by passing through **gates** (neural nets that regulate the flow of information from one time step to the next



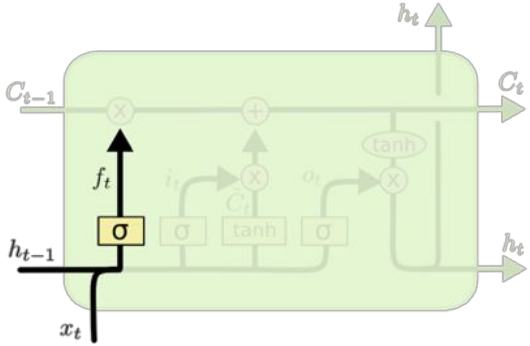



# Gates

- Gates control which information is let through
- They are composed of a sigmoid neural net layer and a pointwise multiplication operation
- The sigmoid layer outputs numbers between 0 and 1, representing how much information should be let through
- 1 = all information, 0 = no information

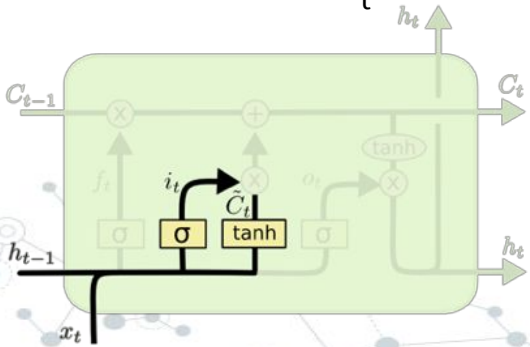


**Step 1: Forget Gate** - Determine how much of the previous state should affect the current state based on the current observed input $x_t$




$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

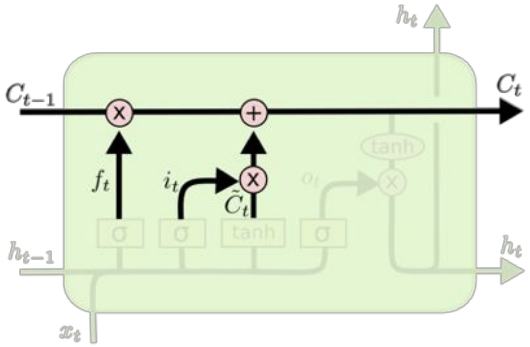
**Step 2: Update Cell State** - First determine which values we will update and by how much (gate $i_t$), then create a list of candidate values that we will add to the current state ($C_t$) based on the current input ($x_t$) and the previous output ($h_{t-1}$).




$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

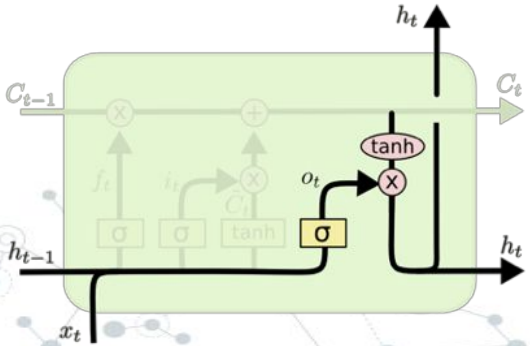
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

**Step 3: Execute the Update** - update the cell state $C_{t-1}$ to $C_t$.




$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

**Step 4: Compute Unit Output** - determine which parts of the cell state will be used as unit output. Output is a filtered version of the cell state.




$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$

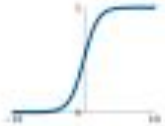
$$h_t = o_t * \tanh\left(C_t\right)$$

# Why tanh?

- To overcome the vanishing/exploding gradient problem
- Forces values to be between -1 and 1

**Sigmoid**

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



**tanh**

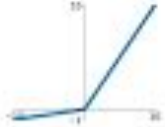
$$\tanh(x)$$

**ReLU**

$$\max(0, x)$$

**Leaky ReLU**

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

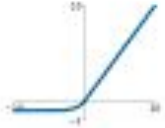


**Maxout**

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

**ELU**

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



# LSTM Variants

- The steps we went through are for the standard, “normal” LSTM
- There are several variations - see blog post link from previous slide
- Encoder-decoder LSTMS led to the emergence of the

  **Attention Mechanism**

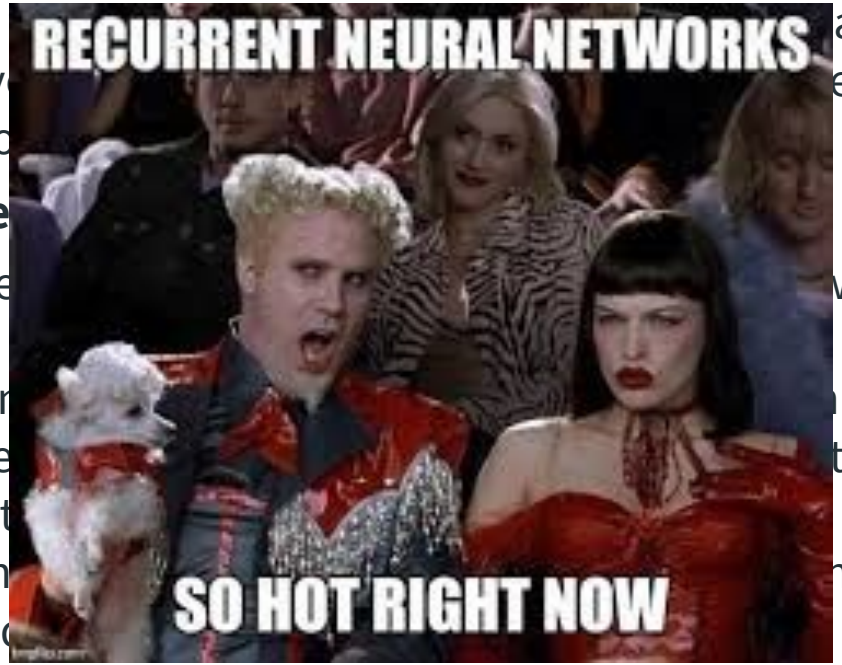
  - Selectively concentrates on a few relevant things while ignoring others
- Think of an encoder as part of a neural net that reads in a sequence, tries to summarize it (encode a context vector), and passes it to the decoder
- The decoder translates the input from the encoder
- The Attention Mechanism overcame shortcomings of encoder-decoder LSTMs and led to huge breakthroughs in NLP

# LSTM Variants

- The steps we al" LSTM
- There are sev evious slide
- Encoder-deco

  **Attention Me**

  - Selective while ignoring others
- Think of an er a sequence, tries to summarize t to the decoder
- The decoder t
- The Attention ncoder-decoder LSTMs and le



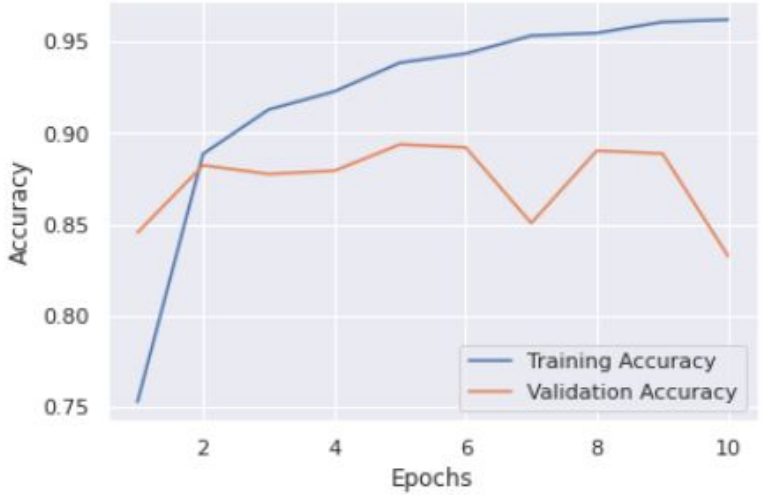
# LSTM in Keras

- Now that you have an idea of how LSTM works, let's implement it in Keras
- We set up a model using an LSTM layer and train it on the IMDb data
- The network is similar to the one with SimpleRNN that we discussed last lecture
- We only specify the output dimensionality of the LSTM layer, and leave every other argument (there's a lot) to the Keras defaults

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Embedding(max_features, 32),

  tf.keras.layers.LSTM(32),

  tf.keras.layers.Dense(1, activation='sigmoid')
])

model.compile(optimizer = tf.keras.optimizers.RMSprop(),
              loss='binary_crossentropy',
              metrics=['accuracy'])

history = model.fit(input_train, y_train,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)
```

# LSTM in Keras

Best performance so far - high 80s in terms of accuracy %

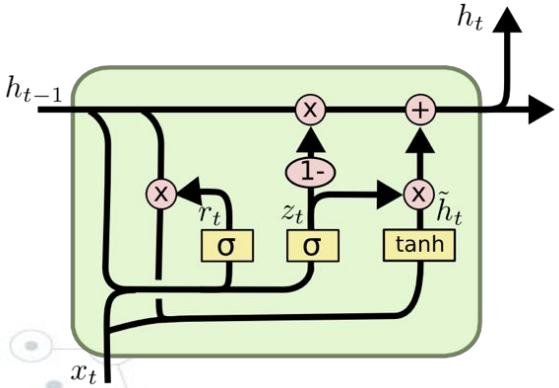



# Gated Recurrent Unit (GRU)

# Gated Recurrent Unit (  )

# GRU

- Relatively new (2014)
- Combines the “forget” and “input” gates into an “update gate”
- Merges cell state and hidden state
- Performance on par with LSTM, but is computationally more efficient (due to fewer tensor operations)




$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

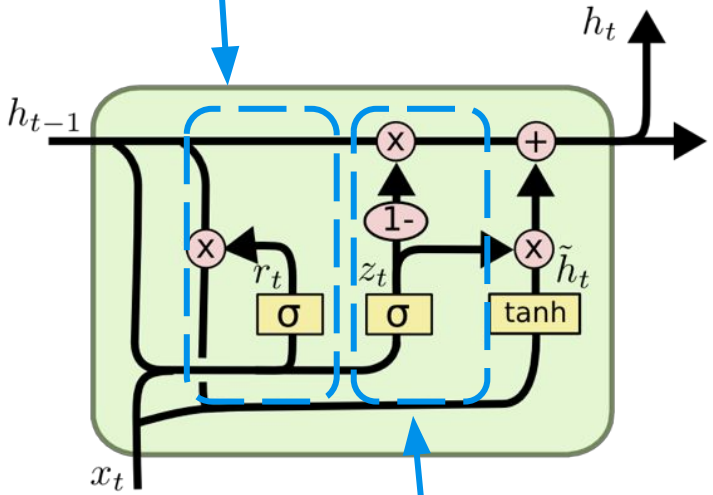
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

# GRU

Reset gate (how much past information to forget)




$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

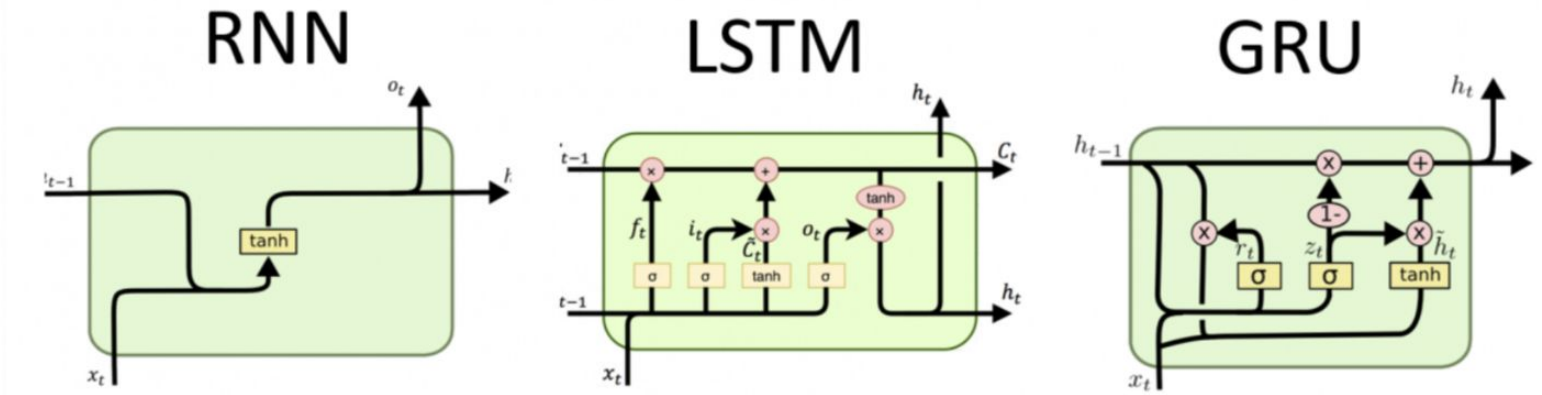
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Update gate (decides which new information to throw away and which to add

# LSTM vs GRU

- LSTM performs better on long sequences
- GRUs are faster to train
- GRUs are simpler to understand and modify
- A nice video explanation/post of LSTM and GRU



# Improving RNN Performance and Generalization

# Improving RNNs

- We will cover 3 techniques for improving RNNs:
  - **Recurrent dropout**: fights overfitting, different from the kind of dropout you are already familiar with
  - **Stacking recurrent layers**: increases generalizability, but comes with a higher computational cost
  - **Bidirectional recurrent layers**: increase accuracy and fight forgetting issues

# Example: temperature forecasting

- RNNs can be applied to any type of sequence data, not just text

- We will be using a **weather timeseries** dataset recorded at the [Weather Station at Max Planck Institute for Biochemistry] in Jena, Germany

# Example: temperature forecasting

- 14 different variables were recorded every 10 minutes over several years, starting in 2003
  - Air temperature, atmospheric pressure, humidity, wind direction, etc.
  - **1 recording every 10 minutes = 6 recordings per hour = 144 recordings per day = 52,560 recordings per year**

- We will be using data from 2009-2016 to build a model that predicts air temperature 24 hours in the future using data from the last few days
- Colab notebook
- Data file

```
fname = 'path/jena_climate_2009_2016.csv'
f = open(fname)
data = f.read()
f.close()

lines = data.split('\n')      # Each line is 1 recording
header = lines[0].split(',') # Variable names are separated by commas
lines = lines[1:]             # Drop first line (it's a header)

print(header)
print(len(lines))
```

```
['"Date Time"', '"p (mbar)"', '"T (degC)"', '"Tpot (K)"', '"Tdew (degC)"', '"rh (%)"',
420551
```

```
import numpy as np

float_data = np.zeros((len(lines), len(header) - 1))
for i, line in enumerate(lines):
    values = [float(x) for x in line.split(',')[1:]]
    float_data[i, :] = values
print(float_data.shape)
```

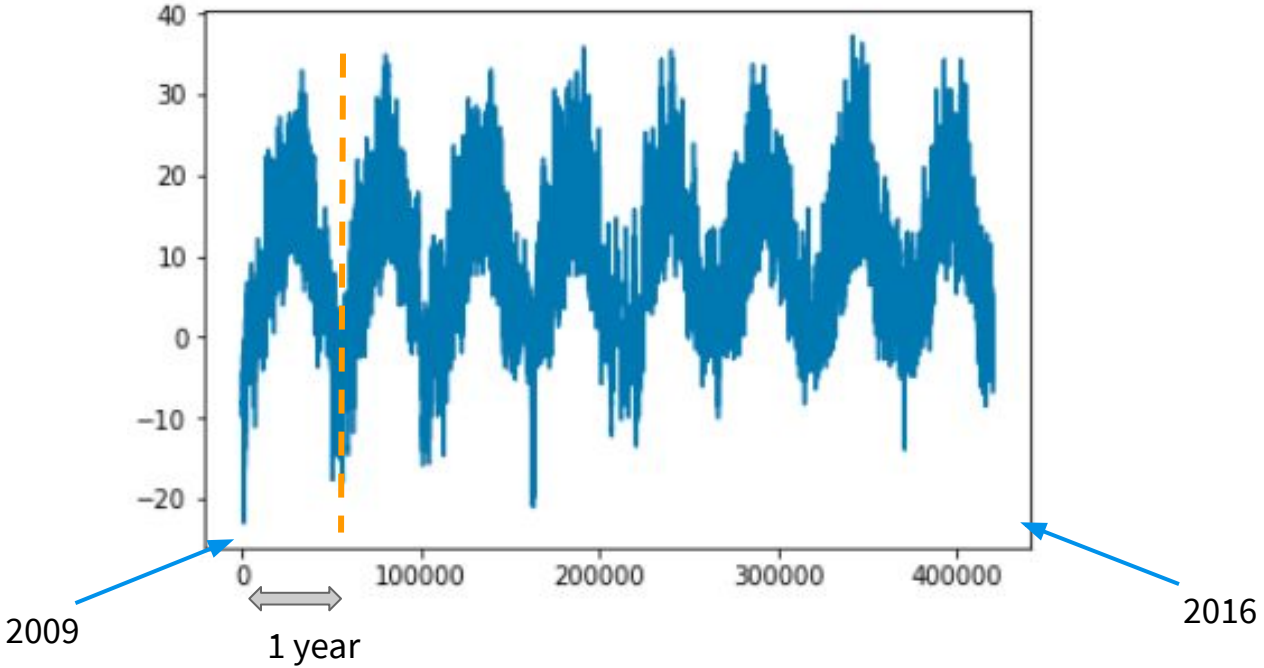
Number of rows (observations)

Number of columns (-1 for unnecessary 1st column: date/time)

Drop first column (the unnecessary date/time)

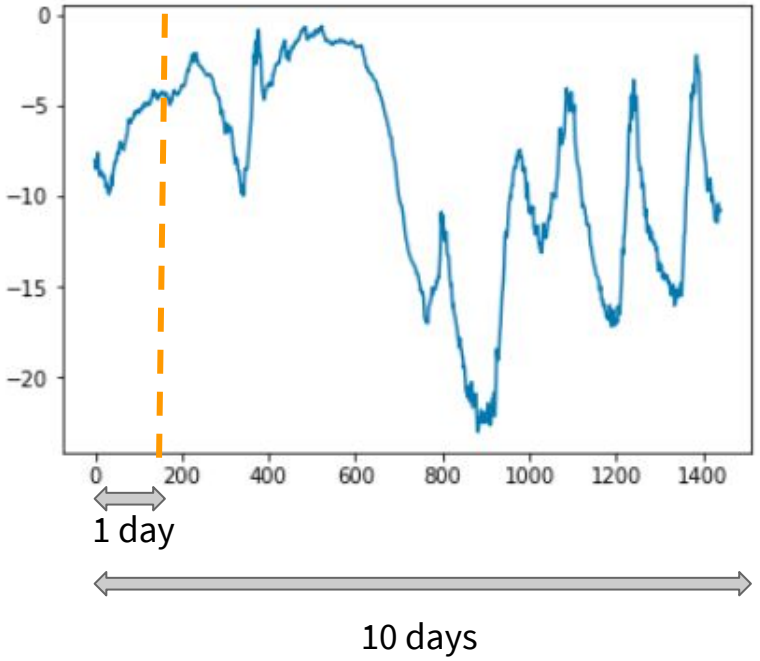
```
(420551, 14)
```

# Temperature over time



# Temperature over time

- Let’s plot the temperature over time (a few days)
- Notice that there is periodicity present, but that it isn’t as consistent as the last plot - this will make predicting the weather in the next 24 hours using data from a few days beforehand more challenging



# Temperature Forecasting

- Task: given data going as far back as **lookback** timesteps (here a timestep is 10 minutes) and sampled every **steps** timesteps, can you predict the temperature in **delay** timesteps?

- **lookback** = 1440; we will go back 10 days
- **steps** = 6; observations will be sampled at one data point per hour - we will only take into account every 6th recording
- **delay** = 144; targets will be 24 hours in the future

- Process the data:
  - Normalize all variables to have mean 0 and standard deviation 1

# Temperature Forecasting in Keras

Generate samples

- **data**: The original array of floating point data
- **lookback**: How many timesteps back should our input data go
- **delay**: How many timesteps in the future should our target be

# Temperature Forecasting in Keras

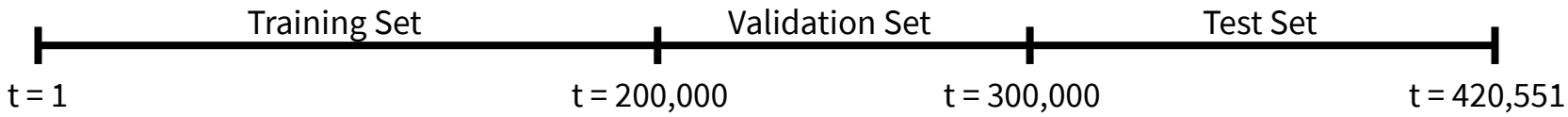
- **min_index** and **max_index**: Indices in the data array that delimit which timesteps to draw from. This is useful for keeping a segment of the data for validation and another one for testing.

- **shuffle**: Whether to shuffle our samples or draw them in chronological order

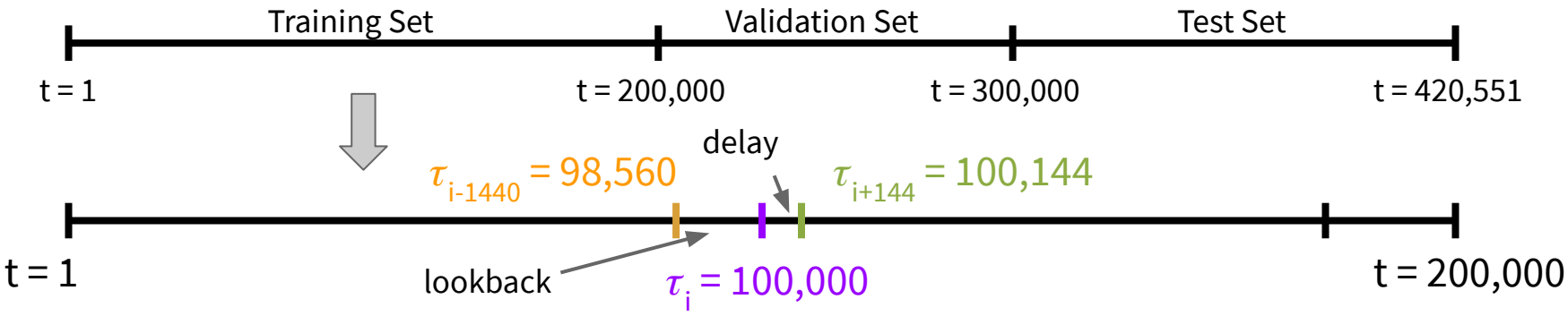
- **batch_size**: The number of samples per batch

- **step**: The period, in timesteps, at which we sample data. We will set it to 6 in order to draw one data point every hour

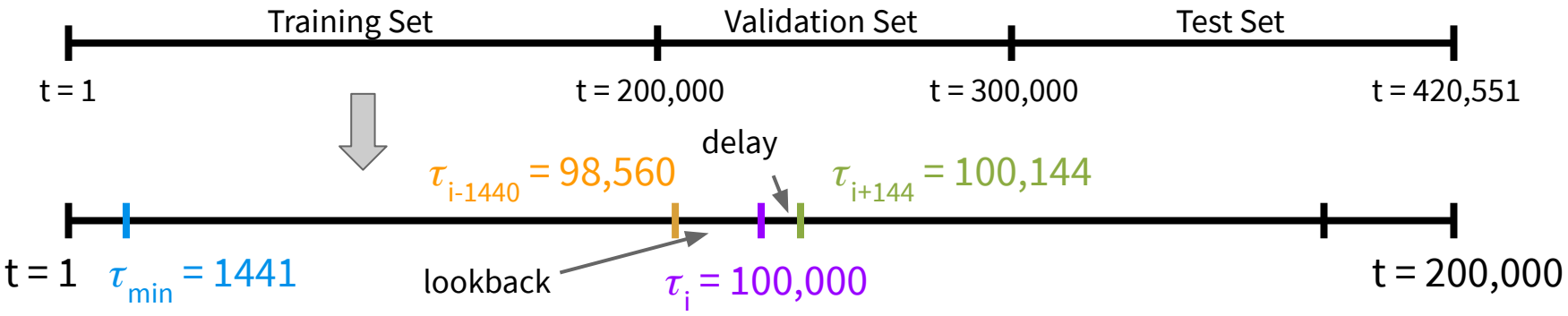
# Timeline


Training Set
Validation Set
Test Set
t = 1
t = 200,000
t = 300,000
t = 420,551

# Timeline

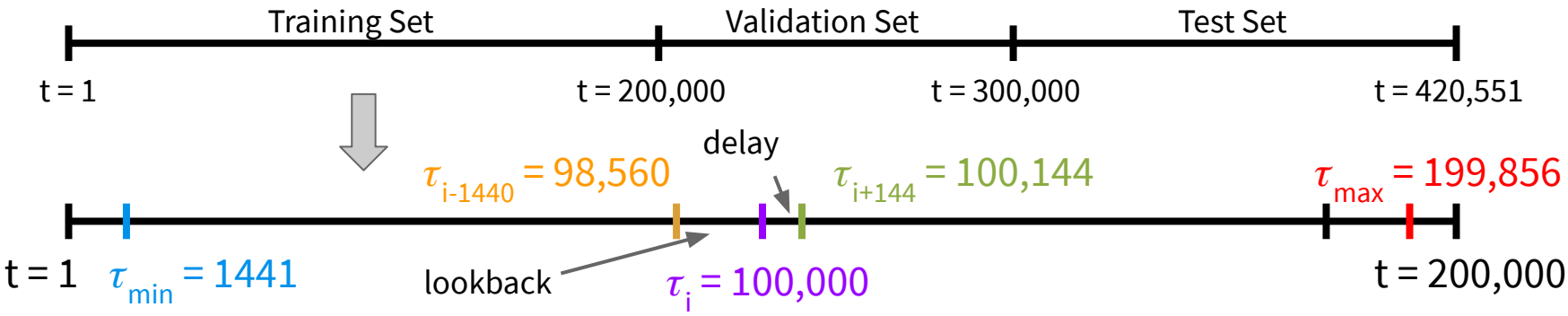

Training Set
Validation Set
Test Set
t = 1
t = 200,000
t = 300,000
t = 420,551
delay
$\tau_{i-1440}$ = 98,560
$\tau_{i+144}$ = 100,144
t = 1
lookback
$\tau_i$ = 100,000
t = 200,000

# Timeline




$\tau_{min}$: we need all 10 days worth of past data to predict the temperature for the next time point. Thus, the minimum value $\tau_i$ can take is 1400 (10 days of previous data) + 1 (time point) = 1441. If we choose $\tau_i$ < 1441, we won't have enough prior data to make a prediction.

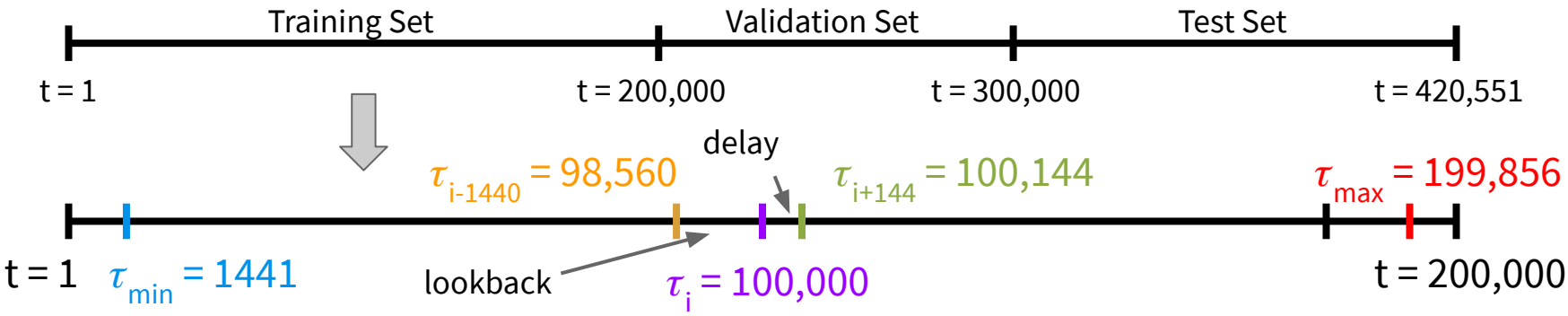
# Timeline




$\tau_{min}$: we need all 10 days worth of past data to predict the temperature for the next time point. Thus, the minimum value $\tau_i$ can take is 1400 (10 days of previous data) + 1 (time point) = 1441. If we choose $\tau_i$ < 1441, we won't have enough prior data to make a prediction.

$\tau_{max}$: we need 24 hours worth of data after this point in order to have a point to make a prediction for. Thus, the maximum value $\tau_i$ can take is 200,000 - 144 = 199,856. If we choose $\tau_i$ >199,856, we won't have a data point to make a prediction for.

# Timeline




$\tau_{min}$: we need all 10 days worth of past data to predict the temperature for the next time point. Thus, the minimum value $\tau_i$ can take is 1400 (10 days of previous data) + 1 (time point) = 1441. If we choose $\tau_i$ < 1441, we won't have enough prior data to make a prediction.

$\tau_{max}$: we need 24 hours worth of data after this point in order to have a point to make a prediction for. Thus, the maximum value $\tau_i$ can take is 200,000 - 144 = 199,856. If we choose $\tau_i$ >199,856, we won't have a data point to make a prediction for.

Steps:

1. Randomly sample a point in time, $\tau_i$, between $\tau_{min}$ and $\tau_{max}$
2. Keep 10 days of data prior to $\tau_i$ and 24 hours after $\tau_i$.
3. Repeat this process multiple times
4. Split training examples into batches
5. Feed into the network
6. Repeat similar process for validation and test sets

Whether to shuffle points in time or not.

While true, meaning while there are still examples to include in a batch

```
def generator(data, lookback, delay, min_index, max_index,
              shuffle=False, batch_size=128, step=6):
    if max_index is None:
        max_index = len(data) - delay - 1
    i = min_index + lookback
    while 1:
        if shuffle:
            rows = np.random.randint(
                min_index + lookback, max_index, size=batch_size)
        else:
            if i + batch_size >= max_index:
                i = min_index + lookback
            rows = np.arange(i, min(i + batch_size, max_index))
            i += len(rows)

        samples = np.zeros((len(rows),
                            lookback // step,
                            data.shape[-1]))
        targets = np.zeros((len(rows),))
        for j, row in enumerate(rows):
            indices = range(rows[j] - lookback, rows[j], step)
            samples[j] = data[indices]
            targets[j] = data[rows[j] + delay][1]
        yield samples, targets
```

For the training set - randomly choose points in time

For the validation and test sets - choose batches of timesteps (in chronological order)

Floor division

One batch of input data

Corresponding target temperatures

Use the generator function to instantiate three generators, one for training, one for validation and one for testing.

Each will look at different temporal segments of the original data: the training generator looks at the first 200,000 timesteps, the validation generator looks at the following 100,000, and the test generator looks at the remainder.

```
lookback = 1440
step = 6
delay = 144
batch_size = 128

train_gen = generator(float_data,
                      lookback=lookback,
                      delay=delay,
                      min_index=0,
                      max_index=200000,
                      shuffle=True,
                      step=step,
                      batch_size=batch_size)
val_gen = generator(float_data,
                    lookback=lookback,
                    delay=delay,
                    min_index=200001,
                    max_index=300000,
                    step=step,
                    batch_size=batch_size)
test_gen = generator(float_data,
                     lookback=lookback,
                     delay=delay,
                     min_index=300001,
                     max_index=None,
                     step=step,
                     batch_size=batch_size)

# This is how many steps to draw from `val_gen`
# in order to see the whole validation set:
val_steps = (300000 - 200001 - lookback) // batch_size

# This is how many steps to draw from `test_gen`
# in order to see the whole test set:
test_steps = (len(float_data) - 300001 - lookback) // batch_size

print(val_steps)
print(test_steps)
```

```
769
930
```

# Temperature Forecasting

- We need to come up with a baseline benchmark to beat
- Common-sense approach: always predict that the temperature 24 hours from now will be equal to the temperature now
- We'll use mean absolute error (MAE) to measure loss

```
def evaluate_naive_method():
    batch_maes = []
    for step in range(val_steps):
        samples, targets = next(val_gen)
        preds = samples[:, -1, 1]
        mae = np.mean(np.abs(preds - targets))
        batch_maes.append(mae)
    print(np.mean(batch_maes))

evaluate_naive_method()
```

```
0.2897359729905486
```

We get MAE = 0.29.

Since our temperature data has been normalized to be centered at 0 and have a standard deviation of 1, this number is not immediately interpretable. It translates to an average absolute error of 0.29 * temperature_std degrees Celsius, i.e. 2.57℃.

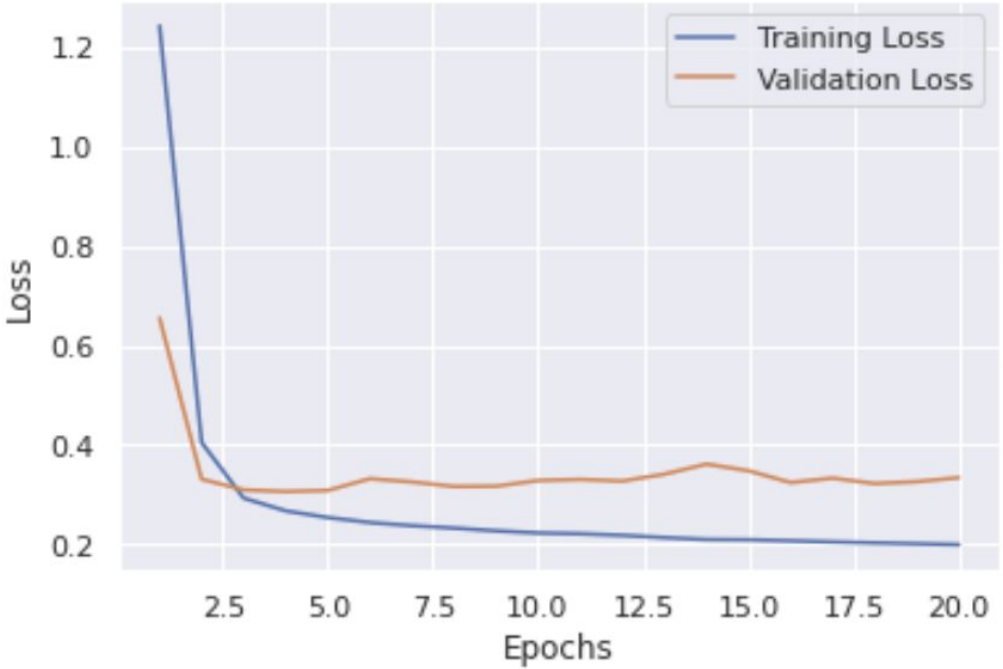
That's a fairly large average absolute error – now the task is to leverage our knowledge of deep learning to do better.

# Temperature Forecasting - Simple Model

- Let's first try a simple model (MLP) before developing a more complex one
- In general it's best to start with a basic model and then work your way up in complexity

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(lookback // step, float_data.shape[-1])),

  tf.keras.layers.Dense(32, activation='relu'),
  tf.keras.layers.Dense(1)
])

model.compile(optimizer = tf.keras.optimizers.RMSprop(),
              loss='mae')

history = model.fit(train_gen,
                    steps_per_epoch=500,
                    epochs=20,
                    validation_data=val_gen,
                    validation_steps=val_steps)
```

# Temperature Forecasting - Simple Model



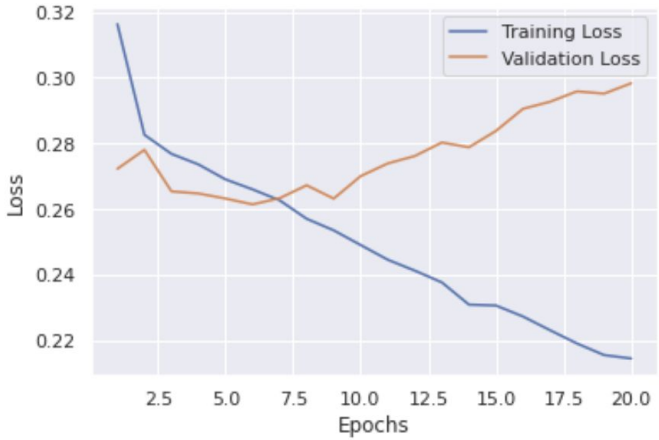

We get MAEs above 0.3 for the validation loss - worse than our benchmark.

This shows that the simple model isn't complex enough for our data and task (it's not taking time into account), and that some benchmarks can be difficult to beat.

# Temperature Forecasting - RNN

This model is better than the previous simple model and the common-sense baseline. There is evidence of overfitting, so let's try dropout next.

```
model = tf.keras.models.Sequential([
  tf.keras.layers.GRU(32, input_shape=(None, float_data.shape[-1])),

  tf.keras.layers.Dense(1)
])

model.compile(optimizer = tf.keras.optimizers.RMSprop(),
              loss='mae')

history = model.fit(train_gen,
                    steps_per_epoch=500,
                    epochs=20,
                    validation_data=val_gen,
                    validation_steps=val_steps)
```



# Recurrent Dropout

# Recurrent Dropout

- It turns out that the classic technique of dropout we saw in earlier lectures can't be applied in the same way for recurrent layers
  - Applying dropout before a recurrent layer impedes learning rather than helping to implement regularization

- The proper way to apply dropout with a recurrent network was discovered in 2015
  - Yarin Gal, "Uncertainty in Deep Learning (PhD Thesis),"
  - **The same pattern of dropped units should be applied at every timestep**

# Recurrent Dropout

- This allows the network to properly propagate its learning error rate through time - a temporally random dropout pattern would disrupt the error signal and hinder the learning process

- Yarin's mechanism has been built into Keras

- Every recurrent layer has 2 dropout-related arguments:
  - **dropout**: a float number specifying the dropout rate for input units of the layer
  - **recurrent_dropout**: a float number specifying the dropout rate of the recurrent units

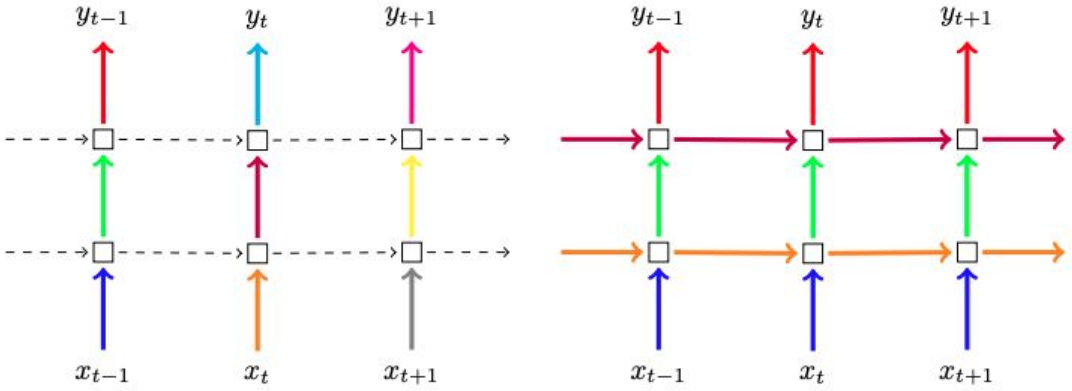
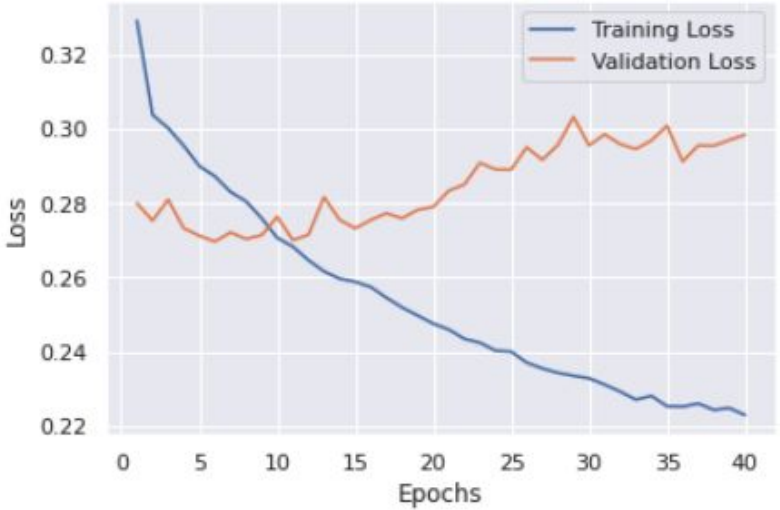



Figure 1: **Depiction of the dropout technique following our Bayesian interpretation (right) compared to the standard technique in the field (left).** Each square represents an RNN unit, with horizontal arrows representing time dependence (recurrent connections). Vertical arrows represent the input and output to each RNN unit. Coloured connections represent dropped-out inputs, with different colours corresponding to different dropout masks. Dashed lines correspond to standard connections with no dropout. Current techniques (naive dropout, left) use different masks at different time steps, with no dropout on the recurrent layers. The proposed technique (Variational RNN, right) uses the same dropout mask at each time step, including the recurrent layers.

Source: https://arxiv.org/pdf/1512.05287.pdf

# Recurrent Dropout in Keras

```
model = tf.keras.models.Sequential([
  tf.keras.layers.GRU(32,
                      dropout = 0.2,
                      recurrent_dropout=0.2,
                      input_shape=(None, float_data.shape[-1])),

  tf.keras.layers.Dense(1)
])

model.compile(optimizer = tf.keras.optimizers.RMSprop(),
              loss='mae')

history = model.fit(train_gen,
                    steps_per_epoch=500,
                    epochs=40,
                    validation_data=val_gen,
                    validation_steps=val_steps)
```




This helps a little with overfitting - increasing the dropout percentage might help more.
We have more stable evaluation scores, but our best scores are not much lower than they were previously

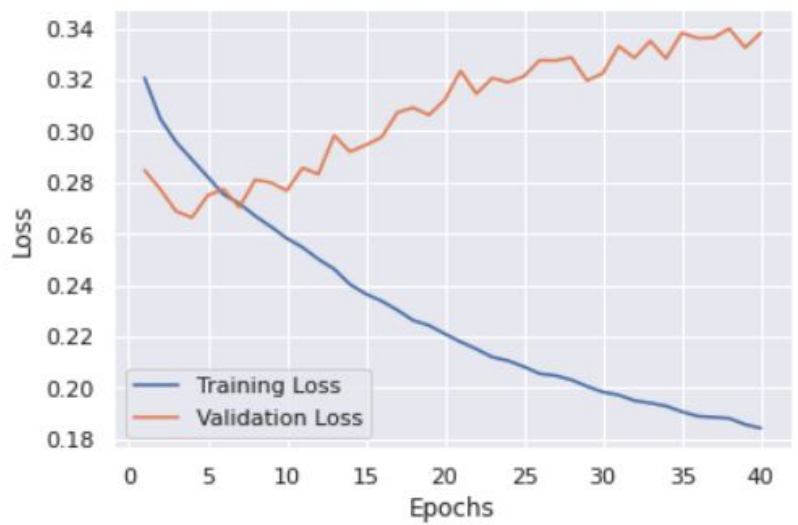
# Stacking Recurrent Layers

# Stacking Recurrent Layers

- Because we have hit a performance bottleneck, we should consider increasing the capacity of the network - make the model more complex
- Increasing network capacity is typically done by increasing the number of units in the layers or adding more layers.
- Recurrent layer stacking is a classic way to build more-powerful recurrent networks: for instance, what currently powers the Google Translate algorithm is a stack of 8 large LSTM layers—that's huge!

# Stacking Recurrent Layers in Keras

```
model = tf.keras.models.Sequential([
  tf.keras.layers.GRU(32,
                      dropout = 0.1,
                      recurrent_dropout=0.5,
                      return_sequences=True,
                      input_shape=(None, float_data.shape[-1])),
  tf.keras.layers.GRU(64, activation='relu',
                      dropout = 0.1,
                      recurrent_dropout=0.5),
  tf.keras.layers.Dense(1)
])

model.compile(optimizer = tf.keras.optimizers.RMSprop(),
              loss='mae')

history = model.fit(train_gen,
                    steps_per_epoch=500,
                    epochs=40,
                    validation_data=val_gen,
                    validation_steps=val_steps)
```

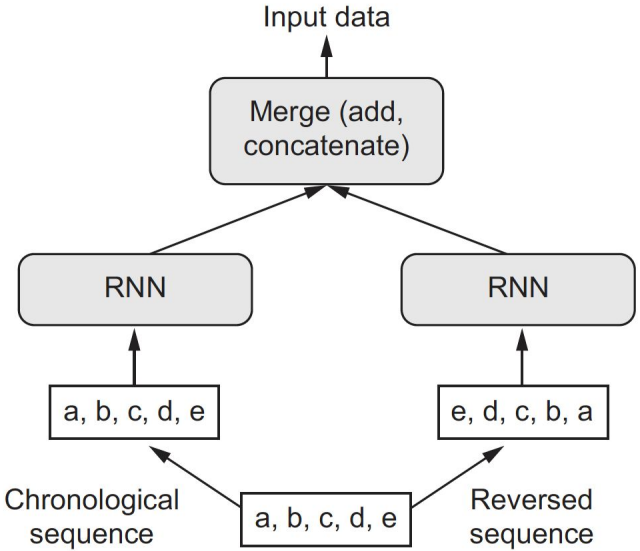



- The overfitting becomes worse, signaling the network capacity is too high, i.e. the model is too complex and has too many parameters.
- It would probably be best to drop the added layer and increase the number of nodes in the first GRU layer

# Bidirectional Recurrent Layers

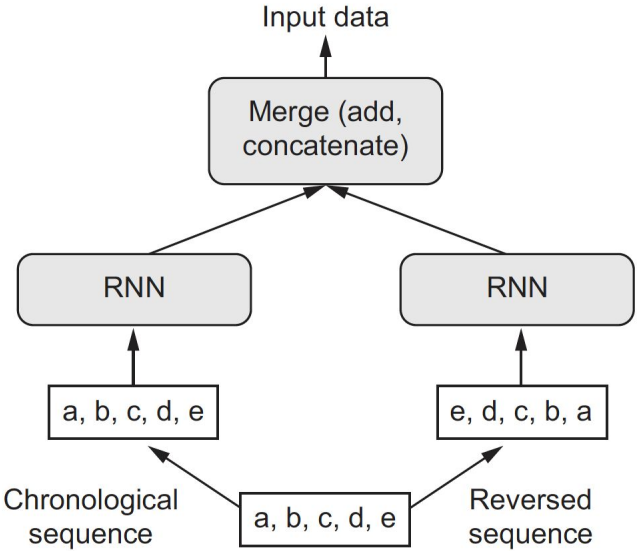
# Bidirectional RNNs

- A bidirectional RNN (BRNN) can offer greater performance on certain tasks
- Frequently used in natural-language processing (NLP)
- BRNNs exploit the order sensitivity of RNNs



# Bidirectional RNNs

- Uses 2 regular RNNs, each of which processes the input sequence in one direction (chronologically and anti chronologically), and then merges their representations
- Catches patterns that may be overlooked by a regular RNN



# Temperature Forecasting with a BRNN

It performs about as well as the regular GRU layer. It's easy to understand why: all of the predictive capacity must be coming from the chronological half of the network, since the anti-chronological half is known to be severely underperforming on this task (again, because the recent past matters much more than the distant past in this case).

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Bidirectional(tf.keras.layers.GRU(32),
                                input_shape=(None, float_data.shape[-1])),

  tf.keras.layers.Dense(1)
])

model.compile(optimizer = tf.keras.optimizers.RMSprop(),
              loss='mae')
history = model.fit(train_gen,
                    steps_per_epoch=500,
                    epochs=40,
                    validation_data=val_gen,
                    validation_steps=val_steps)
```

# Summary

- There are several other things you can try to improve performance
  - Change the number of units in each recurrent layer
  - Try using LSTM layers instead of GRU layers
  - Change the learning rate used by the RMSprop optimizer (or any optimizer)
  - Try a bigger densely connected classifier on top of the recurrent layers