BST 261: Data Science II
Lecture 13

RNNs Continued,
Text Generation

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

Recipe of the Day!

[Dominique Ansel's Peanut Butter Chocolate Crunch Cake]

Paper Presentations

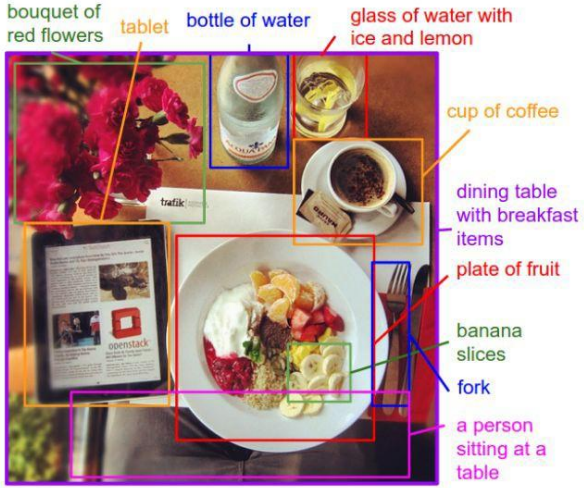
“Deep Visual-Semantic Alignments for Generating Image Descriptions”

Andrej Karpathy, Li Fei-Fei - Department of Computer Science, Stanford University

Presented by Nick Birk - Department of Biostatistics, Harvard University

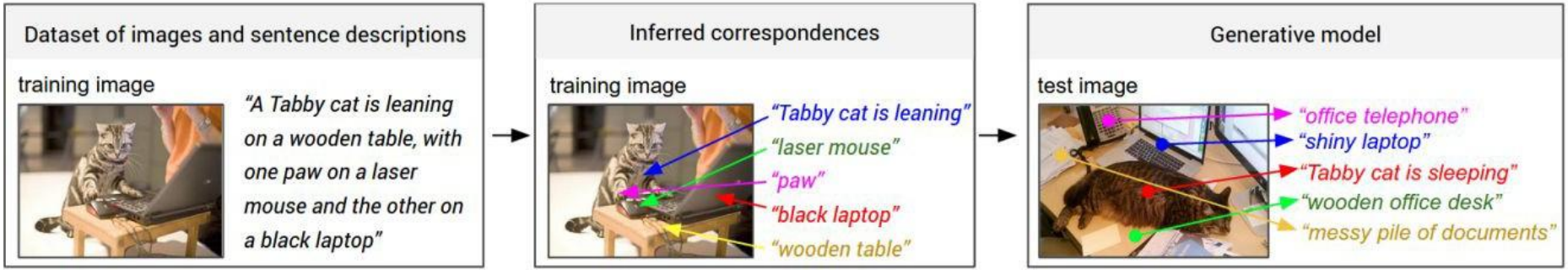
Motivation

- Models to generate image descriptions had been relatively restrictive
- Can we use image captions as a training set to learn to describe the objects in a particular region of an image?
- Focus is on “richer and higher-level descriptions of regions”



Motivation

- Note that this work was performed in 2 stages.
- Stage 1: Images and captions used as training data to make inference about image regions
- Stage 2: Model learns to generate novel descriptions of images



Model - Stage 1

- Objects are detected in each image using a Region Convolutional Neural Network (RCNN) pre-trained on ImageNet
- Each image is represented by a set of h-dimensional vectors
- We use this same h when mapping a set of N words to create a closer correspondence between the words and images.
- Word representations are computed using a Bidirectional Recurrent Neural Network (BRNN)
- The BRNN uses two streams of processing: Left-to-right and right-to-left

Model - Stage 1

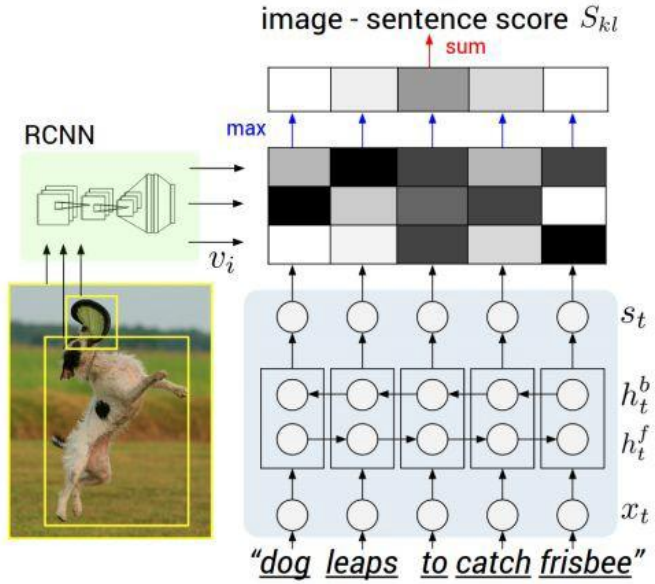
- Ultimately, we utilize a score to match images and sentences. The loss function between image k and sentence l is given by:

$$C(\theta) = \sum_k \Big[\underbrace{\sum_l max(0, S_{kl} - S_{kk} + 1)}_{\text{rank images}} + \underbrace{\sum_l max(0, S_{lk} - S_{kk} + 1)}_{\text{rank sentences}} \Big]. \quad (9)$$

- Each S_{lk} represents the score for the image-sentence pair.
- Score is high if words are supported by image
- Alignments are treated as variables in a Markov Random Field

Model - Stage 1

- Helpful schematic for understanding sentence-image score:

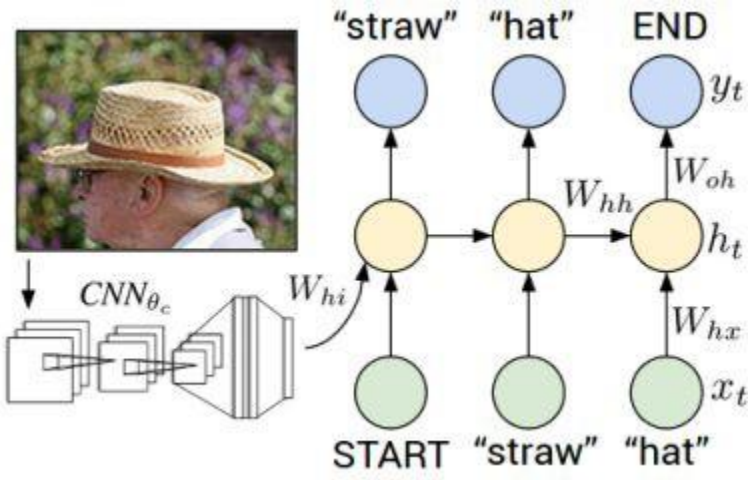


Model - Stage 2

- The Multimodal RNN uses words, context from previous timesteps, and information from image at the first time step to generate a caption.
- This includes a dictionary, with captions beginning at the START token and continuing until the END token.
- Best results were achieved using RMSprop
- The authors note that this was the more difficult part of the model to optimize, largely because of imbalances between common and rare words.

Model - Stage 2

- Schematic for understanding the generative model:



Results

- Images sourced from Flickr8K, Flickr30K, and MSCOCO data sets. Each image has 5 separate sentences as annotation
- Filtered text to only include words which appeared at least 5 times in the training set.

Results - Image-Sentence Alignments

- Able to improve on previous work, authors believe this is in part due to the simpler loss function
- They also note that their regional model helps them to identify specific items within the image like the accordion

Results - Full-image captions

- While many of the captions are reasonable, they are not all correct.
- The current model prioritized speed and simplicity, with a slight decline in performance from other models

man in black shirt is playing guitar.

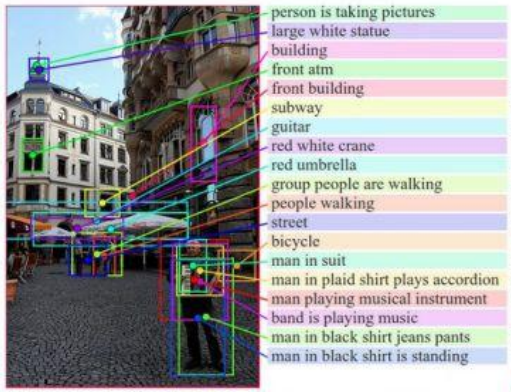
construction worker in orange safety vest is working on road.

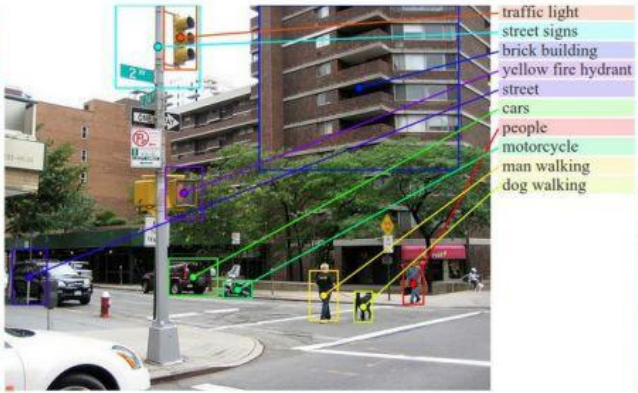
two young girls are playing with lego toy.

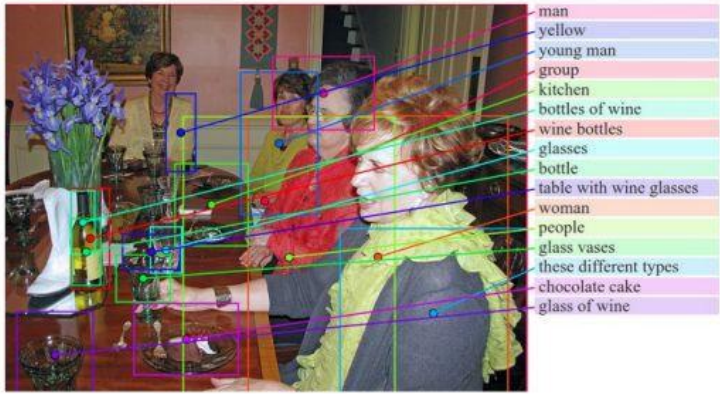
boy is doing backflip on wakeboard.

Results - Regional Captions

- Found that the model outperformed retrieval baselines







Learning to Diagnose with LSTM Recurrent Neural Networks

Zachary C. Lipton, David C. Kale, Charles Elkan, Randall Wetzel

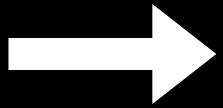
Anna Yang 05/06

Background

Why LSTM?

Data

- ICU clinical data: multivariate time series of observations
- Varying length, irregular sampling, and missing data
- Useful lab results might be separated by days or weeks



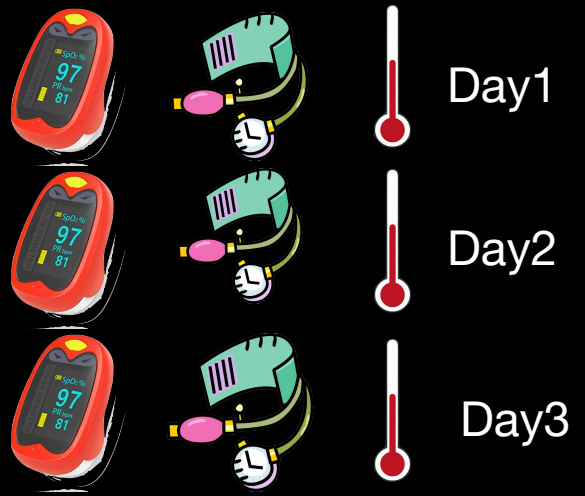
LSTM

- Model varying-length sequential data
- Process nonlinear dynamics
- Capture long range dependencies

ta

Inputs

- 10,401 PICU episodes (12 hours to several months)
- Each a multivariate time series of 13 variables
  - diastolic and systolic bp, peripheral capillary refill rate, end-tidal CO2, fraction of inspired O2, Glascow coma scale, blood glucose, heart rate, pH, respiratory rate, blood oxygen saturation, body temperature, and urine output
- Resample to an hourly rate and take the means
- Imputation for missing values
- Rescale all variables of [0,1]



Data

Outcome

- Each episode: zero or more diagnostic codes (similar to ICD-9) codes
- 429 distinct labels (acute respiratory distress, congestive heart failure, seizures, sepsis)
- Focus on the most common 128 diagnoses
- Multilabel classification based on clinical time series data

Method

Simple RNN model

σ : an element-wise application of the sigmoid (logistic) functio

ϕ : an element-wise application of the tanh function

⊙: Hadamard (element-wise) product.

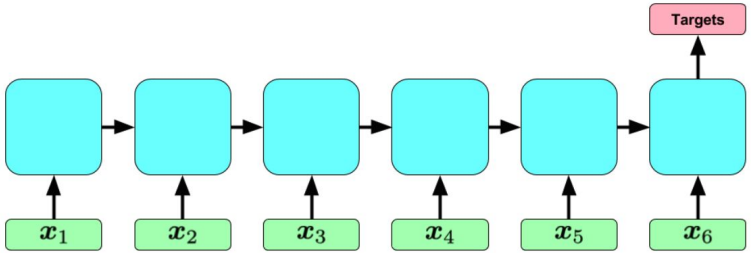
i: input. o: output. f: forget gates

g: input node and has a tanh activation.

$$
\begin{aligned}
\boldsymbol{g}_l^{(t)} &= \phi(W_l^{\mathrm{gx}} \boldsymbol{h}_{l-1}^{(t)} + W_l^{\mathrm{gh}} \boldsymbol{h}_l^{(t-1)} + \boldsymbol{b}_l^{\mathrm{g}}) \\
\boldsymbol{i}_l^{(t)} &= \sigma(W_l^{\mathrm{ix}} \boldsymbol{h}_{l-1}^{(t)} + W_l^{\mathrm{ih}} \boldsymbol{h}_l^{(t-1)} + \boldsymbol{b}_l^{\mathrm{i}}) \\
\boldsymbol{f}_l^{(t)} &= \sigma(W_l^{\mathrm{fx}} \boldsymbol{h}_{l-1}^{(t)} + W_l^{\mathrm{fh}} \boldsymbol{h}_l^{(t-1)} + \boldsymbol{b}_l^{\mathrm{f}}) \\
\boldsymbol{o}_l^{(t)} &= \sigma(W_l^{\mathrm{ox}} \boldsymbol{h}_{l-1}^{(t)} + W_l^{\mathrm{oh}} \boldsymbol{h}_l^{(t-1)} + \boldsymbol{b}_l^{\mathrm{o}}) \\
\boldsymbol{s}_l^{(t)} &= \boldsymbol{g}_l^{(t)} \odot \boldsymbol{i}_l^{(i)} + \boldsymbol{s}_l^{(t-1)} \odot \boldsymbol{f}_l^{(t)} \\
\boldsymbol{h}_l^{(t)} &= \phi(\boldsymbol{s}_l^{(t)}) \odot \boldsymbol{o}_l^{(t)}.
\end{aligned}
$$

Loss function: the average of the losses at each output node

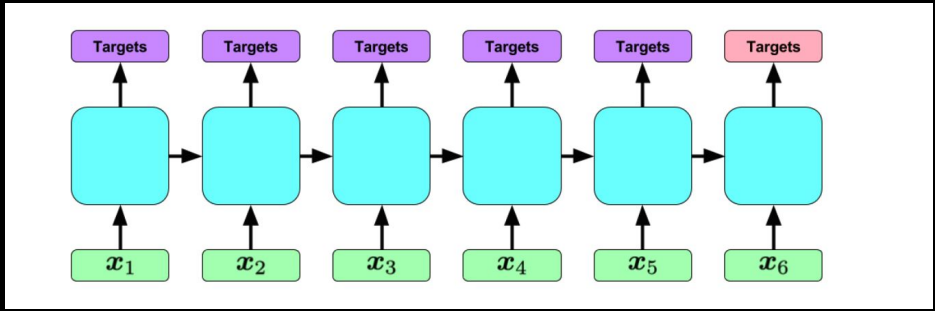
$$\mathrm{loss}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \frac{1}{|L|} \sum_{l=1}^{l=|L|} -(y_l \cdot \log(\hat{y}_l) + (1 - y_l) \cdot \log(1 - \hat{y}_l)).$$



Method (Improve performance)

Sequential Target Replication

- Primary target: Red (Prediction)
- Intermediate targets: purple (Back propagates errors)



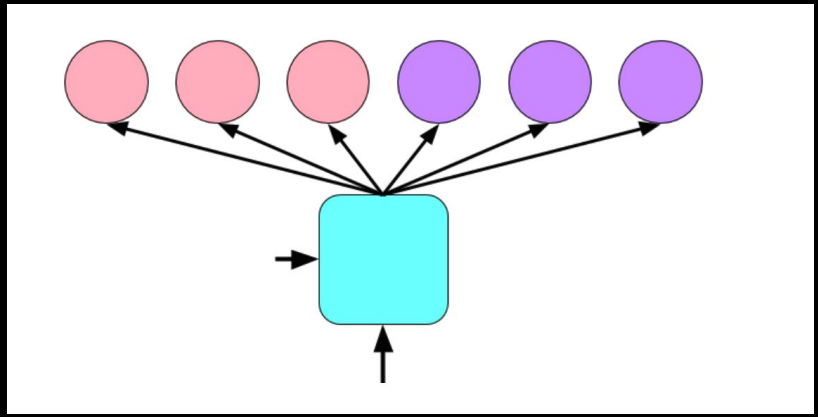
- A convex combination of the loss at the final sequence step and the mean of the losses over all sequence steps

$$\alpha \cdot \frac{1}{T} \sum_{t=1}^{T} \text{loss}(\hat{\boldsymbol{y}}^{(t)}, \boldsymbol{y}^{(t)}) + (1 - \alpha) \cdot \text{loss}(\hat{\boldsymbol{y}}^{(T)}, \boldsymbol{y}^{(T)})$$

Method (Improve performance)

Auxiliary Output Training

- Remaining 301 labels and other information
- Reduce overfitting
- Minimize the loss



Overfitting

Regularization

- Complexity of models and scale of data
- Weight decay
- Dropout to non-recurrent connections

Baseline Classifiers

MLP

- Base rate model (minimum performance baseline)
- Logistic regression
- MLP model
  - 3 hidden layers of 300 hidden units each, rectified linear activations and dropout of 0.5
  - 1000 epochs
  - Input 1: raw time series
  - Inpu2: hand-engineered features (143 features that capture many of the indicators clinicians look for in critical illness)

Results

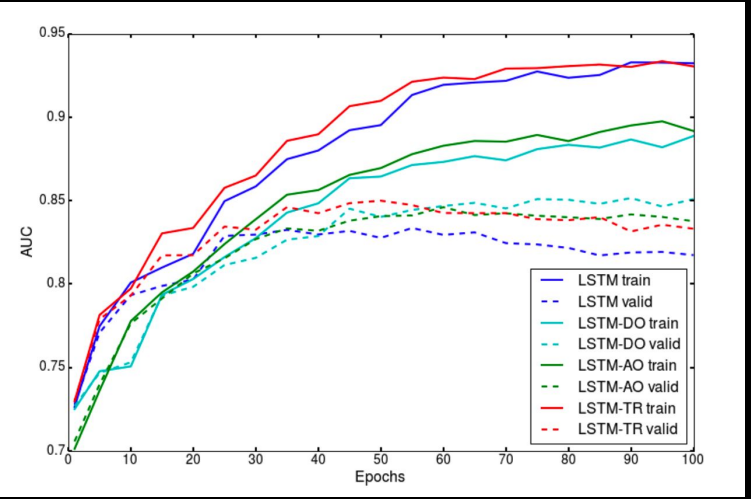
- Best Performing LSTM (LSTM-DO-TR): two layers of 128 memory cells, dropout of probability 0.5 between layers
- Outperforms MLP with hand-engineered features

Classification performance for 128 ICU phenotypes

Model	Micro AUC	Macro AUC	Micro F1	Macro F1	Prec. at 10
Base Rate	0.7128	0.5	0.1346	0.0343	0.0788
Log. Reg., First 6 + Last 6	0.8122	0.7404	0.2324	0.1081	0.1016
Log. Reg., Expert features	0.8285	0.7644	0.2502	0.1373	0.1087
MLP, First 6 + Last 6	0.8375	0.7770	0.2698	0.1286	0.1096
MLP, Expert features	**0.8551**	**0.8030**	**0.2930**	**0.1475**	**0.1170**
LSTM Models with two 64-cell hidden layers					
LSTM	0.8241	0.7573	0.2450	0.1170	0.1047
LSTM, AuxOut (Diagnoses)	0.8351	0.7746	0.2627	0.1309	0.1110
LSTM-AO (Categories)	0.8382	0.7748	0.2651	0.1351	0.1099
LSTM-TR	0.8429	0.7870	0.2702	0.1348	0.1115
LSTM-TR-AO (Diagnoses)	0.8391	0.7866	0.2599	0.1317	0.1085
LSTM-TR-AO (Categories)	0.8439	0.7860	0.2774	0.1330	0.1138
LSTM Models with Dropout (probability 0.5) and two 128-cell hidden layers					
LSTM-DO	0.8377	0.7741	0.2748	0.1371	0.1110
LSTM-DO-AO (Diagnoses)	0.8365	0.7785	0.2581	0.1366	0.1104
LSTM-DO-AO (Categories)	0.8399	0.7783	0.2804	0.1361	0.1123
LSTM-DO-TR	**0.8560**	**0.8075**	**0.2938**	0.1485	0.**1172**
LSTM-DO-TR-AO (Diagnoses)	0.8470	0.7929	0.2735	0.1488	0.1149
LSTM-DO-TR-AO (Categories)	0.8543	0.8015	0.2887	0.1446	0.1161
LSTM-DO-TR (Linear Gain)	0.8480	0.7986	0.2896	**0.1530**	0.1160
Ensembles of Best MLP and Best LSTM					
Mean of LSTM-DO-TR & MLP	0.8611	0.8143	0.2981	0.1553	0.1201
Max of LSTM-DO-TR & MLP	**0.8643**	**0.8194**	**0.3035**	**0.1571**	**0.1218**

Discussion

- LSTM RNNs (TR) outperforms MLP
- Regularization, auxiliary outputs and dropout all reduce the generalization gap
- Indicator for missing values
- Additional clinical experts to assign diagnoses given only 13 variables



Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks

Zhu, Park, Isola, and Efros: UC Berkeley

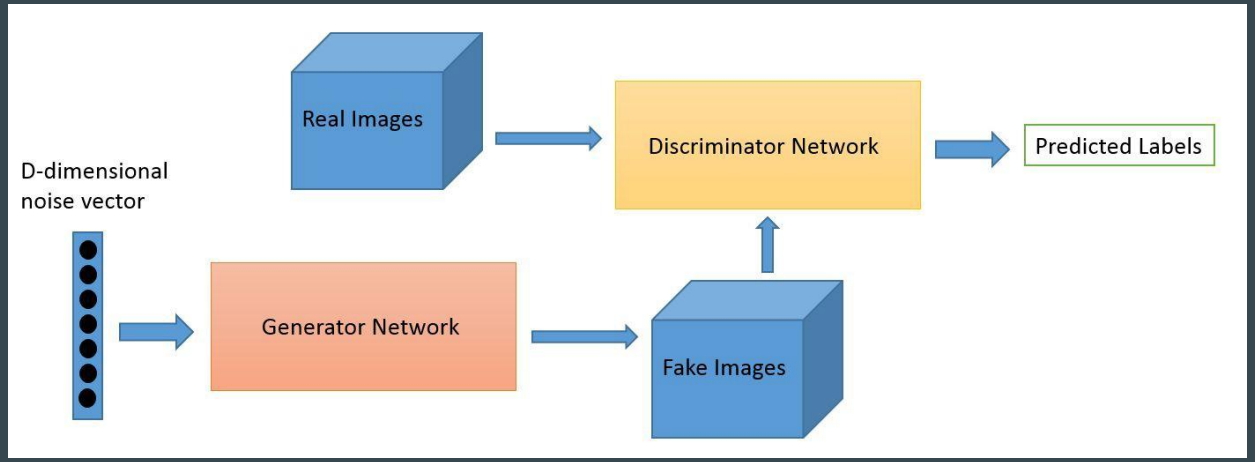
Presented by Daniel Shinnick

Strategy Overview

- Aims to use Generative Adversarial Networks (GANs) to modify images
  - Styles, textures, colors, etc.
- Unpaired images
- Steered towards inverse functions via cyclical structure
- Can be used for style transfer, object transformation, photo enhancement, etc.

GANs Basics

- Robot Artists
- Fake News
- Deepfakes



GANs Learn by Outsmarting Themselves

- Generator tries to fool discriminator
- Process is working when the discriminator starts to think that your “fake images” are “real images”
- Adversarial loss
- $G(X) \rightarrow Y$

Unpaired

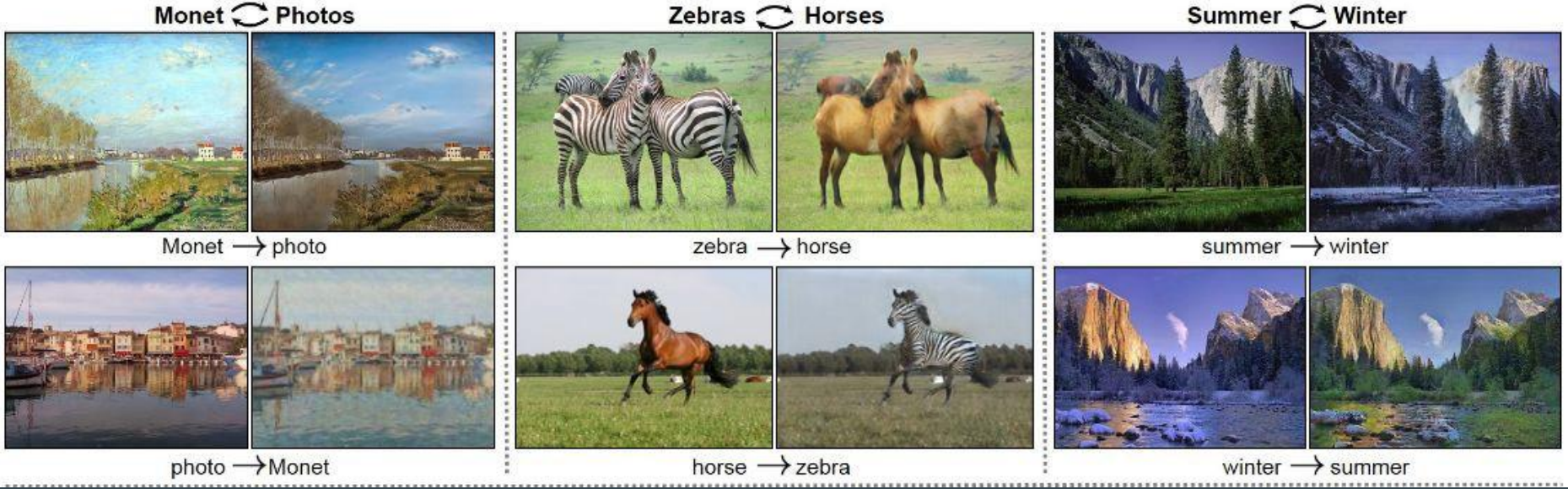
- Many previous models use paired images to learn
  - Paired images are difficult to obtain in more complex settings
  - For object transfiguration, output not well-defined
- Instead they use unpaired X and Y and must ‘translate’ to it

Loss	Map → Photo % Turkers labeled *real*	Photo → Map % Turkers labeled *real*
CoGAN [28]	0.6% ± 0.5%	0.9% ± 0.5%
BiGAN/ALI [7, 6]	2.1% ± 1.0%	1.9% ± 0.9%
Pixel loss + GAN [42]	0.7% ± 0.5%	2.6% ± 1.1%
Feature loss + GAN	1.2% ± 0.6%	0.3% ± 0.2%
CycleGAN (ours)	**26.8% ± 2.8%**	**23.2% ± 3.4%**

Cyclical

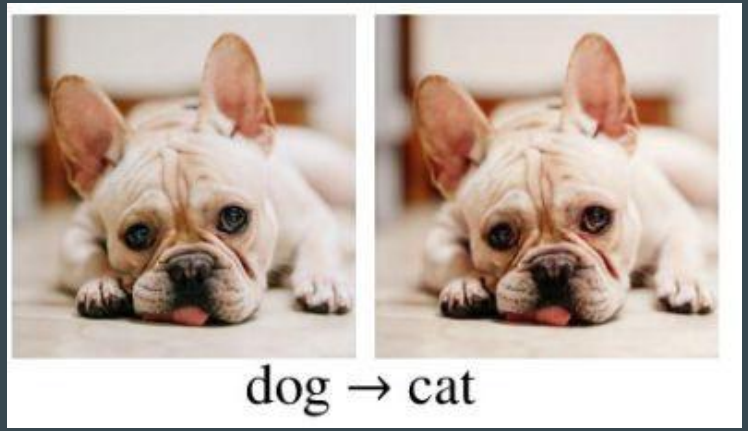
- Translate from X to Y using G, and from Y to X using F
- Cycle consistency loss encourages $F(G(x)) \approx x$ and $G(F(y)) \approx y$
- Combined cyclical and adversarial loss

Successful Examples



Authors cite that they had many examples that did not work as well as these examples

Failed and Outside Results



Putin → Zebra Putin
Video → GTA, Owen → Ramen, MRI →

CT https://junyanz.github.io/CycleGAN/

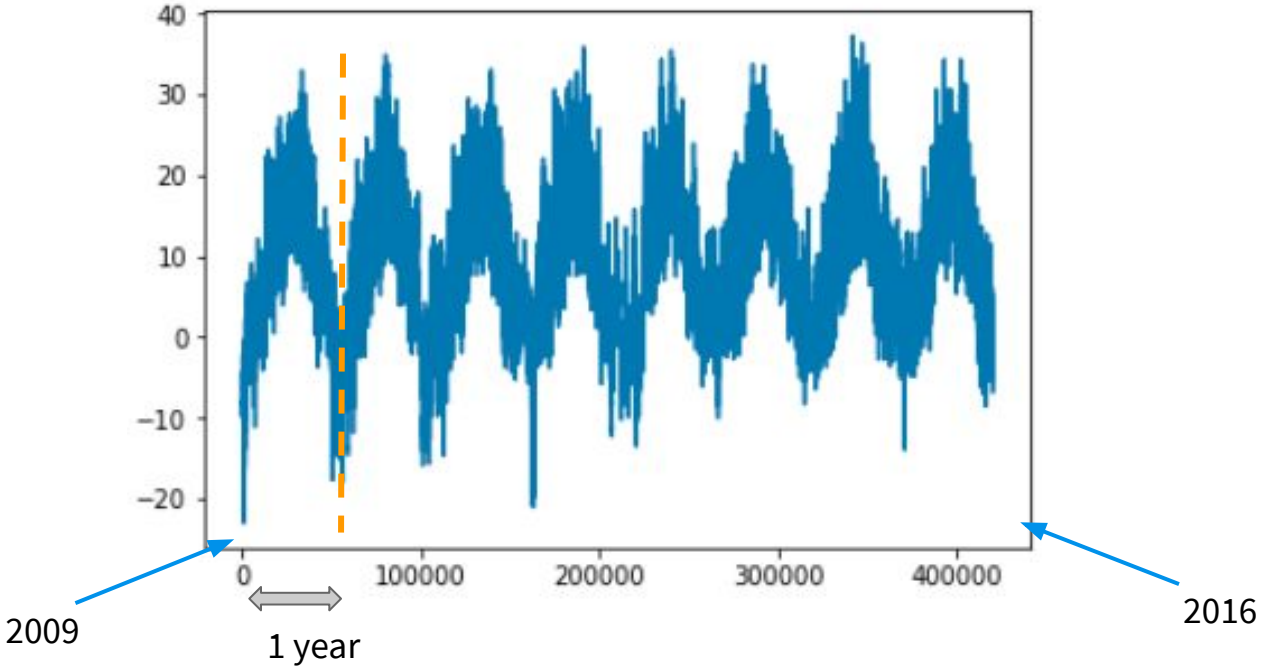
RNNs for Time Series

Example: temperature forecasting

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

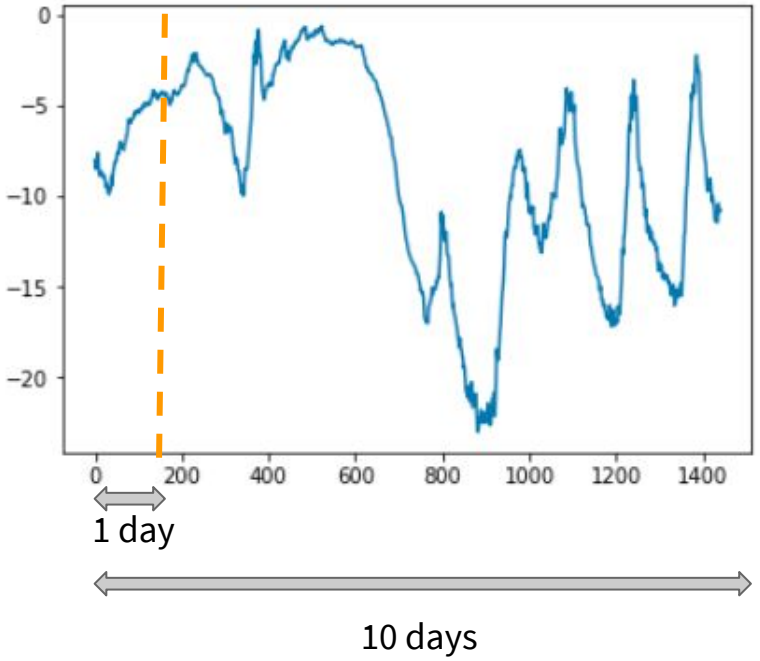
- We will be using a **weather timeseries** dataset recorded at the Weather Station at Max Planck Institute for Biochemistry in Jena, Germany

Temperature over time

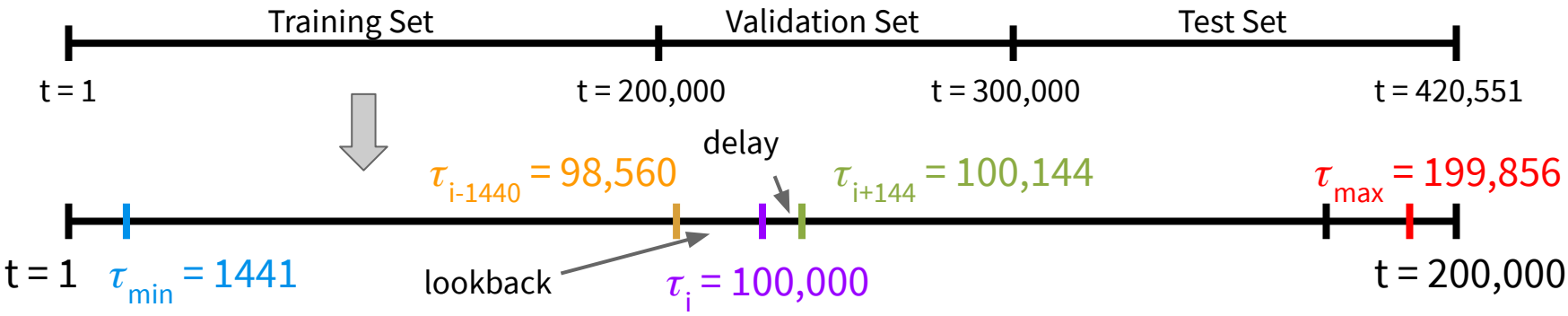


Temperature over time

- 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



Timeline



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

τ_{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 τ_i can take is 200,000 - 144 = 199,856. If we choose τ_i >199,856, we won't have a data point to make a prediction for.

Steps:

1. Randomly sample a point in time, τ_i, between τ_{min} and τ_{max}
2. Keep 10 days of data prior to τ_i and 24 hours after τ_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

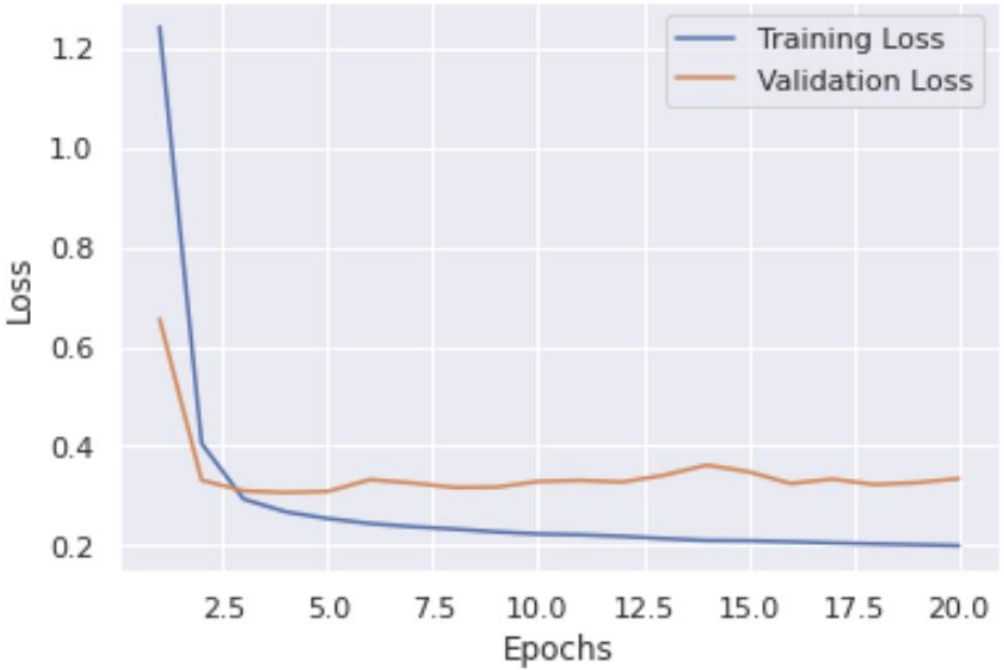
```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(lookback // step, float_data.shape[-1])),

  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

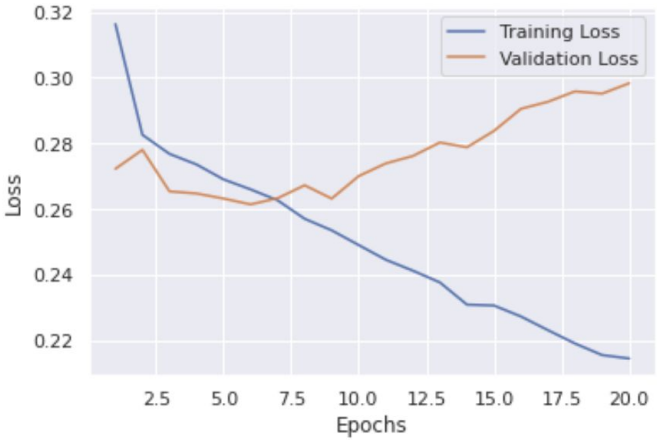
This model is better than the previous simple model and the common-sense baseline. There is evidence of overfitting, so let's try dropout next.

```
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

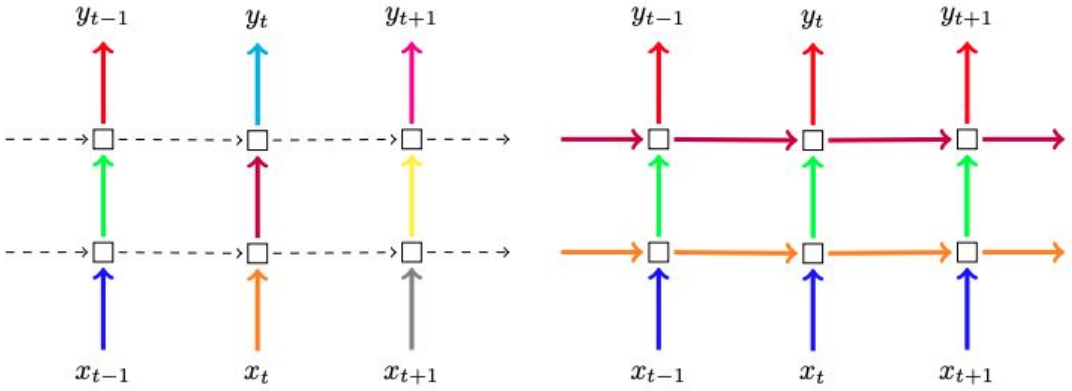
- The proper way to apply dropout with a recurrent network was discovered in 2015
  - Yarin Gal, "Uncertainty in Deep Learning (PhD Thesis),"
  - **The same pattern of dropped units should be applied at every timestep**

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



(a) Naive dropout RNN (b) Variational RNN

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

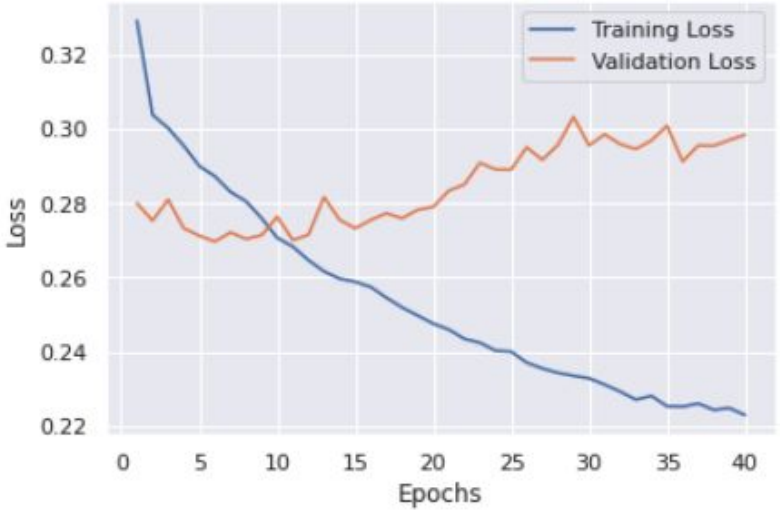
Recurrent Dropout in Keras

```
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

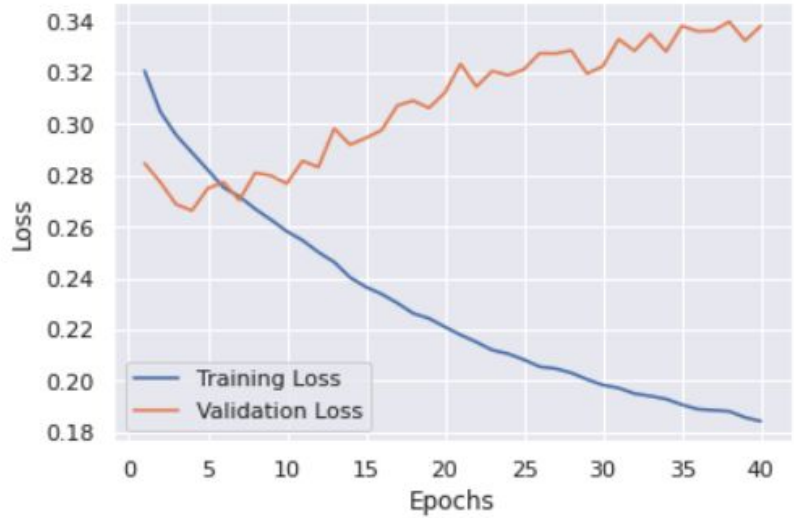
- Because we have hit a performance bottleneck, we should consider increasing the capacity of the network - make the model more complex
- Increasing network capacity is typically done by increasing the number of units in the layers or adding more layers.
- Recurrent layer stacking is a classic way to 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)
```

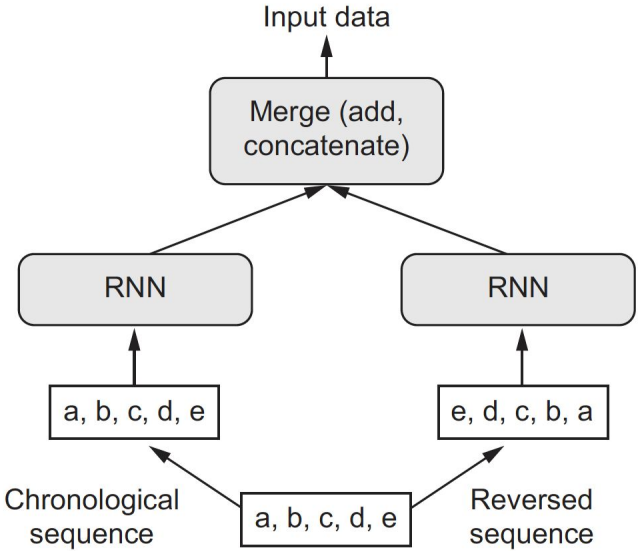


- The overfitting becomes worse, signaling the network capacity is too high, i.e. the model is too complex and has too many parameters.
- It would probably be best to drop the added layer and increase the number of nodes in the first GRU layer

Bidirectional Recurrent Layers

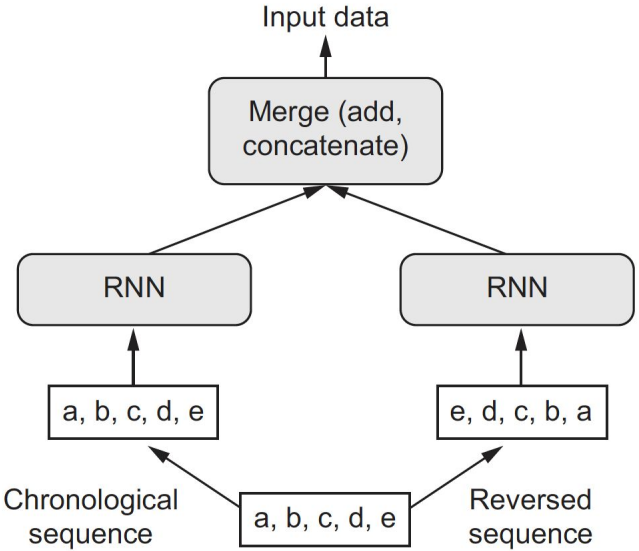
Bidirectional RNNs

- 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

1D Convolution for Sequence Data

1D Convolution for Sequence Data

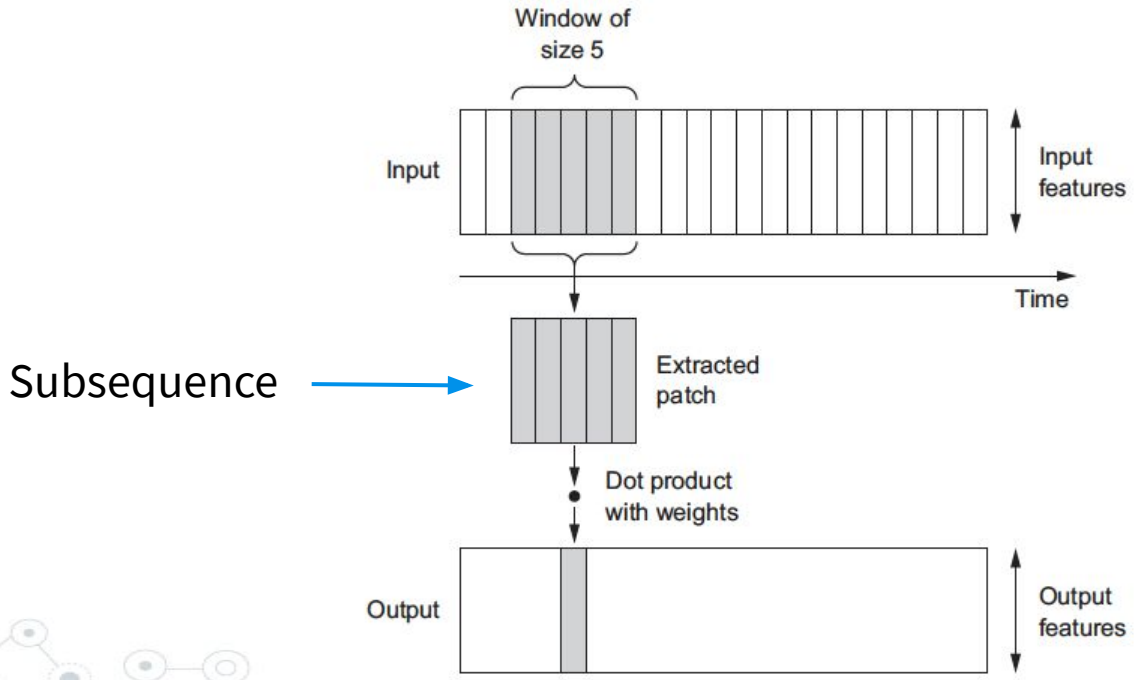
- Recall that CNNs can extract features from local input patches and then recognize them anywhere
- If we think of time as a spatial dimension, we can use 1D CNNs for sequence data
- Great for audio generation and machine translation
- Faster to run than RNNs

1D Convolution for Sequence Data

- **1D CNNs extract subsequences (patches)** from sequences and perform the same transformation on each subsequence
  - A pattern learned at a specific position of a sequence can be recognized at a different position (translation invariant)

- Pooling in this case is similar to what we have seen before - output the maximum or average value of a subsequence

1D Convolution for Sequence Data



1D Convolution for Sequence Data

- In Keras, use the **Conv1D** layer
  - Takes as input (samples, time, features)
  - Returns 3D tensors

- Combine Conv1D layer with a **MaxPooling1D** layer

- End with a **GlobalMaxPooling** or **Flatten** layer

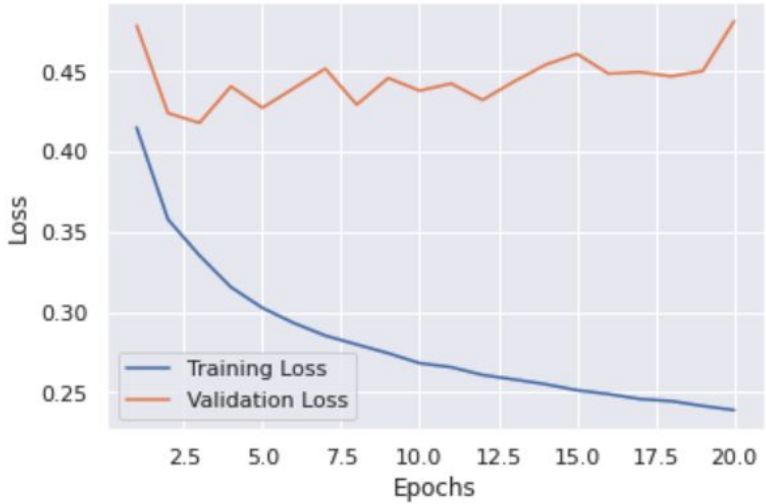
- Can use larger windows for 1D CNNs - typically windows of size 7 or 9
  - In a 2D convolution layer, a 3 x 3 filter contains 9 feature vectors
  - A 1D convolution layer with a window of size 3 contains only 3 vectors

Stacking 1D CNN layers

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Conv1D(32, 5, activation='relu',
                         input_shape=(None, float_data.shape[-1])),
  tf.keras.layers.MaxPooling1D(3),
  tf.keras.layers.Conv1D(32, 5, activation='relu'),
  tf.keras.layers.MaxPooling1D(3),
  tf.keras.layers.Conv1D(32, 5, activation='relu'),
  tf.keras.layers.GlobalMaxPooling1D(),
  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)
```

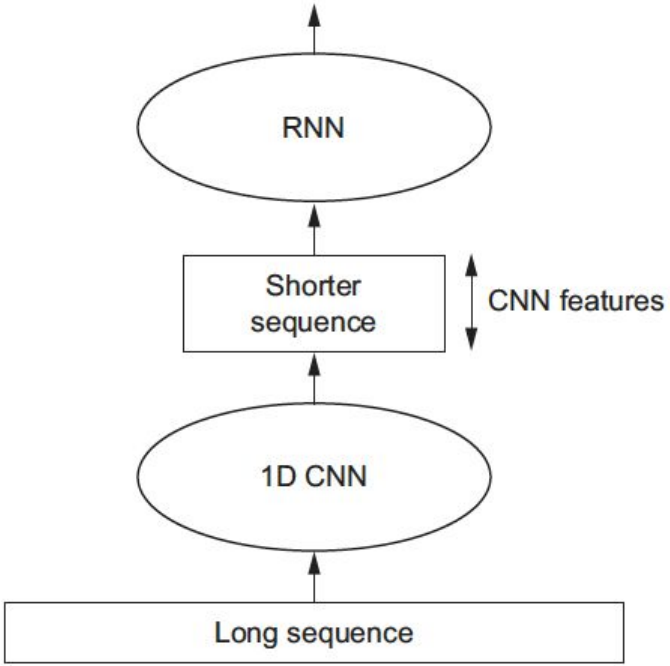


Stacking 1D CNN layers

- 1D CNNs process subsequences independently and aren't sensitive to the order of the timesteps - thus, they don't perform well when faced with long sequences
- Could try stacking 1D CNN layers, but still doesn't induce order sensitivity
- Doesn't beat the common-sense baseline
  - Due to the fact that 1D CNNs aren't time sensitive
- Let's try combining CNNs and RNNs

Combining CNNs and RNNs

- To combine the speed of 1D CNNs with the time sensitivity of RNNs, could do the following:
  - Use the 1D CNN as a preprocessing step
  - Feed this output into an RNN
- The CNN turns long sequences into much shorter sequences



Temperature Forecasting

- With the combination of a CNN and RNN we can now either look at data from longer ago (increase the lookback parameter), or look at high-resolution timeseries (decrease the step parameter)

- Let’s decrease the step parameter by half - this gives us sequences that are twice as long

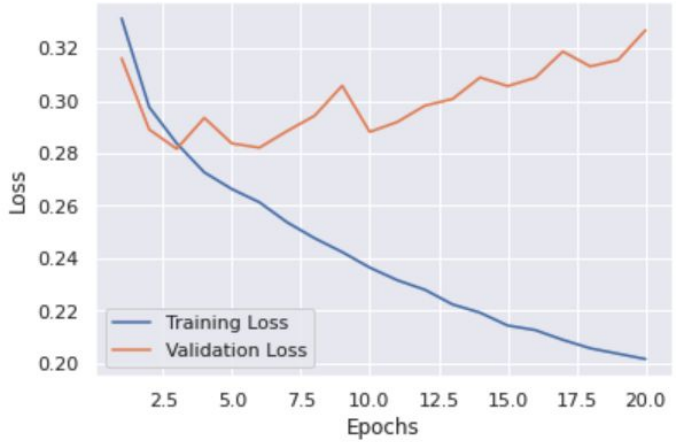
- Temperature data is now sampled at a rate of 1 point per 30 minutes

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Conv1D(32, 5, activation='relu',
                         input_shape=(None, float_data.shape[-1])),
  tf.keras.layers.MaxPooling1D(3),
  tf.keras.layers.Conv1D(32, 5, activation='relu'),
  tf.keras.layers.GRU(32, dropout=0.1, recurrent_dropout=0.5),
  tf.keras.layers.Dense(1)
])

model.summary()

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)
```



This model doesn't perform as well as the regularized GRU model, but it does run a lot faster.

This model also looks at twice as much data as the other model, but that doesn't seem to help with accuracy in this case - it could for other data sets.

Summary

- In the same way that 2D convnets perform well for processing visual patterns in 2D space, 1D convnets perform well for processing temporal patterns. They offer a faster alternative to RNNs on some problems, in particular natural language processing tasks.

- Typically, 1D convnets are structured much like their 2D equivalents from the world of computer vision: they consist of stacks of Conv1D layers and Max-Pooling1D layers, ending in a global pooling operation or flattening operation.

Summary

- Because RNNs are extremely expensive for processing very long sequences, but 1D convnets are cheap, it can be a good idea to use a 1D convnet as a preprocessing step before an RNN, shortening the sequence and extracting useful representations for the RNN to process.

Text Generation with LSTM

Generative Deep Learning

- Generative deep learning methods using RNNs and CNNs have been around for awhile, but have recently been getting a lot of attention
  - 2002: Douglas Eck applied LSTM to music generation - he is now at Google Brain and started a research group called Magenta to use deep learning to create engaging music
  - 2013: Alex Graves applies recurrent mixture density networks to generate human-like handwriting
  - Many more, a one that we'll talk about today

- Many researchers in this field have said that "generating sequential data is the closest computers get to **dreaming**"

Generative RNNs for Text

- RNNs have been successfully used for
  - Music generation
  - Dialogue generation
  - Image generation
  - Speech synthesis
  - Molecule design

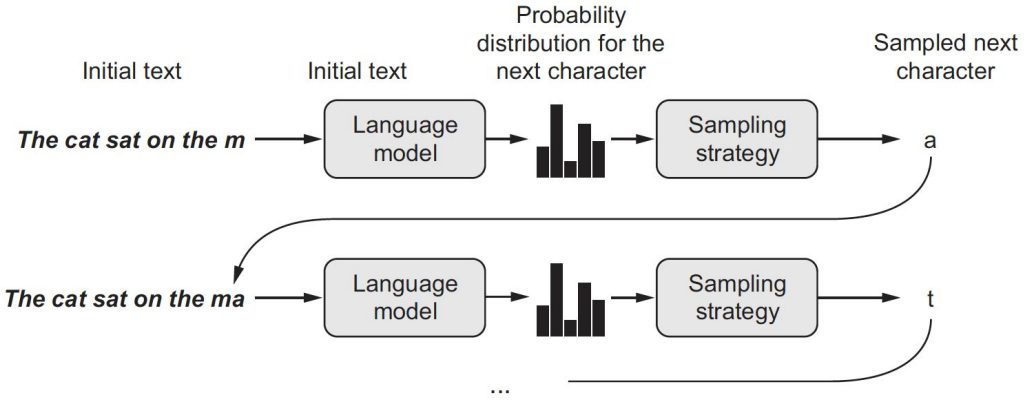
- Main idea for text generation: train a model to predict the next token or next few tokens in a sequence

Generative RNNs for Text

- **Language Model:** any network that can model the probability of the next token given the previous ones
  - Captures the latent space of language - its statistical structure
  - Once it is trained, you can sample from it to generate new sequences

Generative RNNs for Text

- Process
  - 1. Feed it an initial string of text (called conditioning data)
  - 2. Ask the model to generate the next character or word
  - 3. Add the generated output back to the input data
  - 4. Repeat many times



Generative RNNs for Text

- We can choose the next character or word in different ways - some are better than others

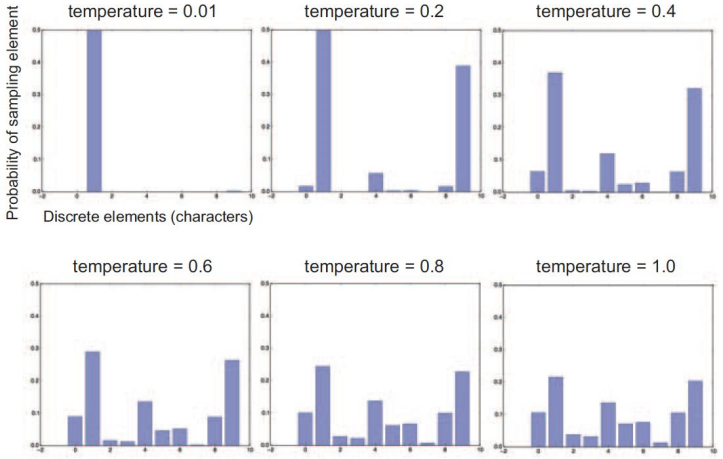
- A naive approach is greedy sampling - always choosing the most likely next character or word
  - This results in repetitive, predictable strings and not very coherent language

Generative RNNs for Text

- Better approach is stochastic sampling
  - Sample next characters or words with specific probability from a probability distribution
  - Allows even unlikely characters or words to be sampled at times, generating more interesting and creative sentences
  - Doesn’t offer a way of controlling the randomness in the sampling process

Generative RNNs for Text

- New parameter to tune: **softmax temperature**
  - Controls the amount of randomness
  - More randomness = similar probability for every character or word and results in more interesting output
  - Less randomness = higher probability for just one or a few characters or words and results in repetitive output
  - Can change the amount of randomness via the temperature value
    - **Higher** temperature = more **randomness**
    - **Lower** temperature = more **deterministic**



Character-level LSTM Generation

- Need a lot of data to train from
- Can choose from many sources, referred to as a **corpus**
  - Wikipedia
  - The Lord of the Rings
  - The writings of Nietzsche translated into English
- Let’s see an example with the writings of Nietzsche as our corpus

Character-level LSTM Generation

- Training the language model and sampling from it:
- Given a trained model and a seed text snippet, do the following repeatedly

    - 1. Draw from the model a probability distribution for the next character, given the generated text available so far
    - 2. Reweight the distribution to a certain temperature
    - 3. Sample the next character at random according to the reweighted distribution
    - 4. Add the new character at the end of the available text

Character-level LSTM Generation

- Random seed:
  - “new faculty, and the jubilation reached its climax when kant"
- Output at epoch 20 with temperature = 0.2:

“new faculty, and the jubilation reached its climax when kant and such a man in the same time the spirit of the surely and the such the such as a man is the sunligh and subject the present to the superiority of the special pain the most man and strange the subjection of the special conscience the special and nature and such men the subjection of the special men, the most surely the subjection of the special intellect of the subjection of the same things and”

Character-level LSTM Generation

- Random seed:
  - “new faculty, and the jubilation reached its climax when kant"
- Output at epoch 20 with temperature = 0.5:

“new faculty, and the jubilation reached its climax when kant in the eterned and such man as it’s also become himself the condition of the experience of off the basis the superiory and the special morty of the strength, in the langus, as which the same time life and "even who discless the mankind, with a subject and fact all you have to be the stand and lave no comes a troveration of the man and surely the conscience the superiority, and when one must be w "

Character-level LSTM Generation

- Random seed:
  - “new faculty, and the jubilation reached its climax when kant"
- Output at epoch 20 with temperature = 1.0:

“new faculty, and the jubilation reached its climax when kant, as a periliting of manner to all definites and transpects it it so hicable and ont him artiar resull too such as if ever the proping to makes as cnecience. to been juden, all every could coldiciousnike hother aw passife, the plies like which might thiod was account, indifferent germin, that everythery certain destrution, intellect into the deteriorablen origin of moralian, and a lessority o"

Character-level LSTM Generation

- Random seed:
  - “new faculty, and the jubilation reached its climax when kant"
- Output at epoch 60 (model has now fully converged) with temperature = 0.2:

“cheerfulness, friendliness and kindness of a heart are the sense of the spirit is a man with the sense of the sense of the world of the self-end and self-concerning the subjection of the strengthorixes–the subjection of the subjection of the subjection of the self-concerning the feelings in the superiority in the subjection of the subjection of the spirit isn’t to be a man of the sense of the subjection and said to the strength of the sense of the"

Character-level LSTM Generation

- Random seed:
  - “new faculty, and the jubilation reached its climax when kant"
- Output at epoch 60 with temperature = 0.5:

“cheerfulness, friendliness and kindness of a heart are the part of the soul who have been the art of the philosophers, and which the one won’t say, which is it the higher the and with religion of the frences. the life of the spirit among the most continuess of the strengther of the sense the conscience of men of precisely before enough presumption, and can mankind, and something the conceptions, the subjection of the sense and suffering and the"

Character-level LSTM Generation

- Random seed:
  - “new faculty, and the jubilation reached its climax when kant"
- Output at epoch 60 with temperature = 1.0:

“cheerfulness, friendliness and kindness of a heart are spiritual by the ciuture for the entalled is, he astraged, or errors to our you idstood–and it needs, to think by spars to whole the amvives of the newoatly, prefectly raals! it was name, for example but voludd atu-especity"–or rank onee, or even all "solett increessic of the world and implussional tragedy experience, transf, or insiderar,–must hast if desires of the strubction is be stronges"

Character-level LSTM Generation

- Low temperature results in repetitive and predictable text, but local structure is highly realistic
- Higher temperatures result in more interesting, surprising and creative text, sometimes creating new words - but the local structure breaks down and most words are strings of random characters
- Generally, somewhere in the middle (around 0.5) creates the most interesting text - but this depends on the corpus and the human reading the results