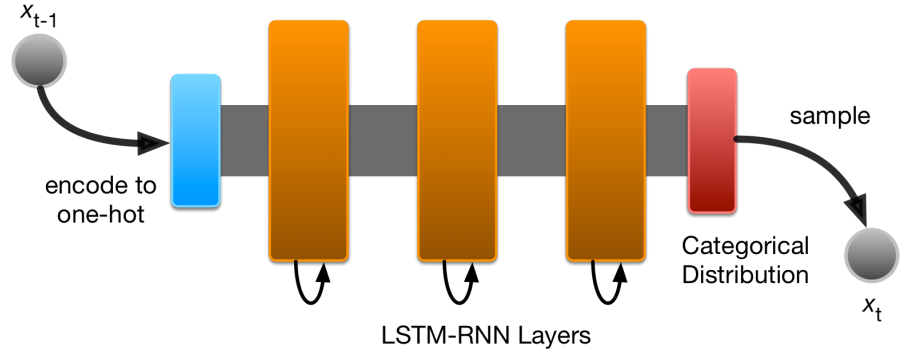
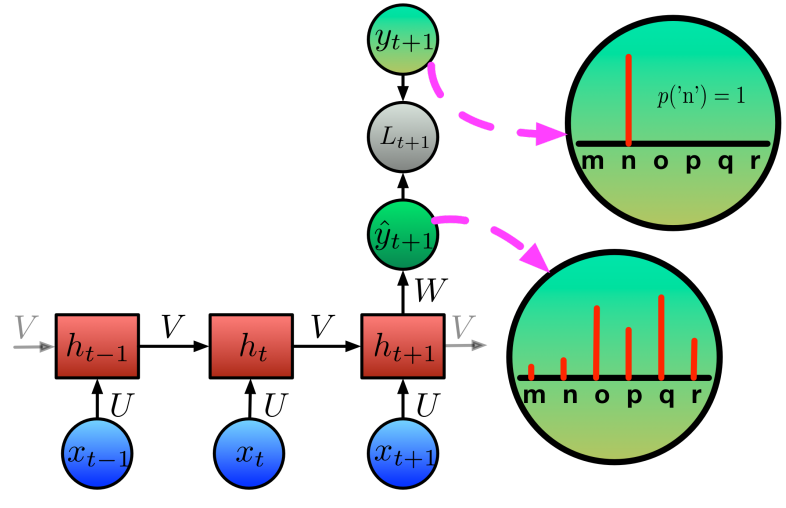
MIXTURE DENSITY NETWORKS

Charles Martin

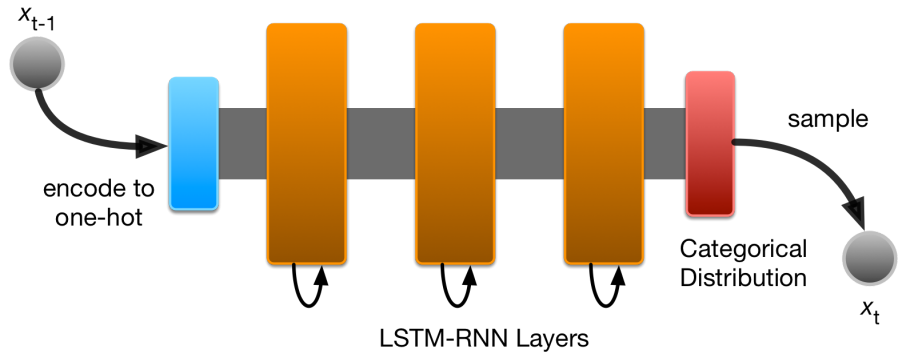
SO FAR; RNNS THAT MODEL CATEGORICAL DATA

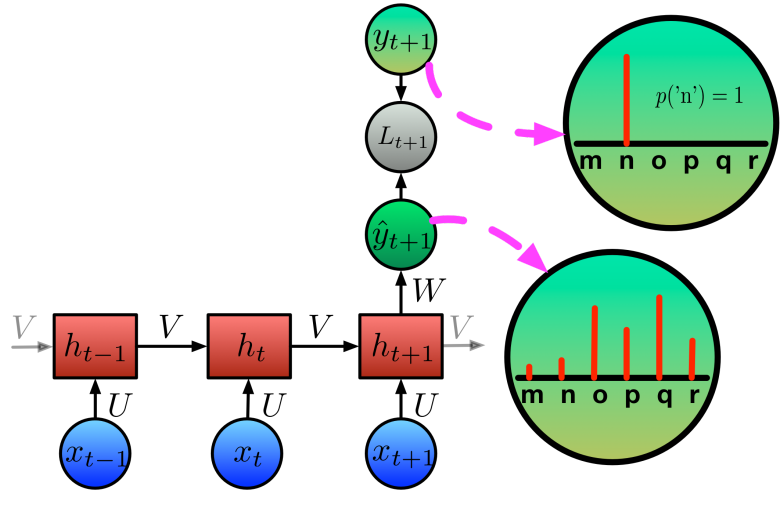




SO FAR; RNNS THAT MODEL CATEGORICAL DATA

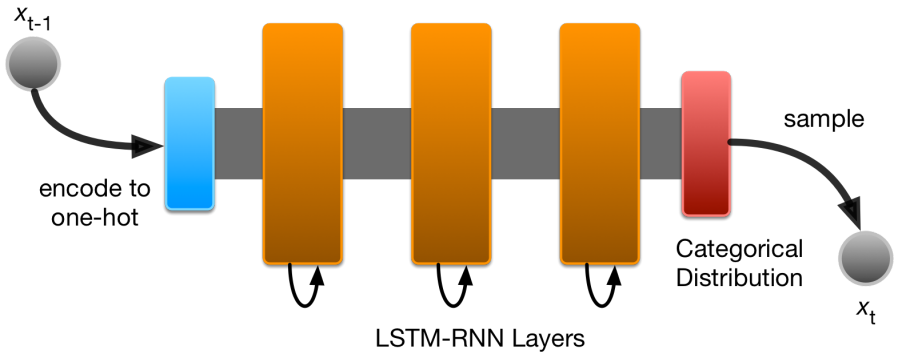
- Remember that most RNNs (and most deep learning models) end with a softmax layer.

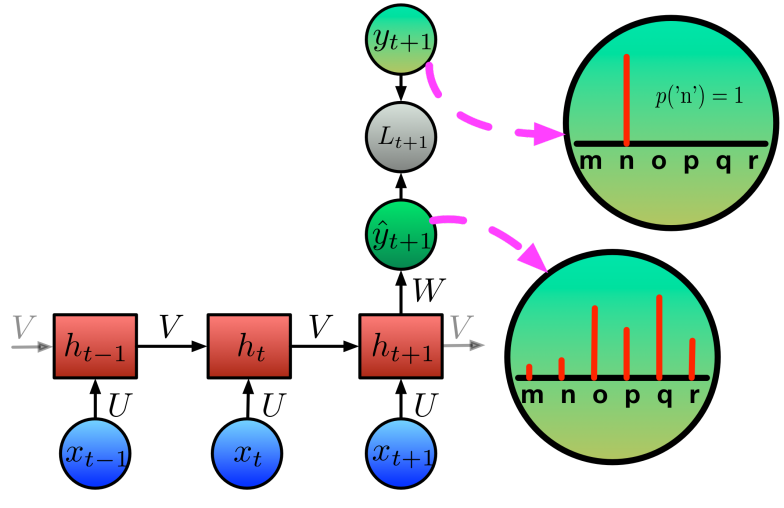




SO FAR; RNNS THAT MODEL CATEGORICAL DATA

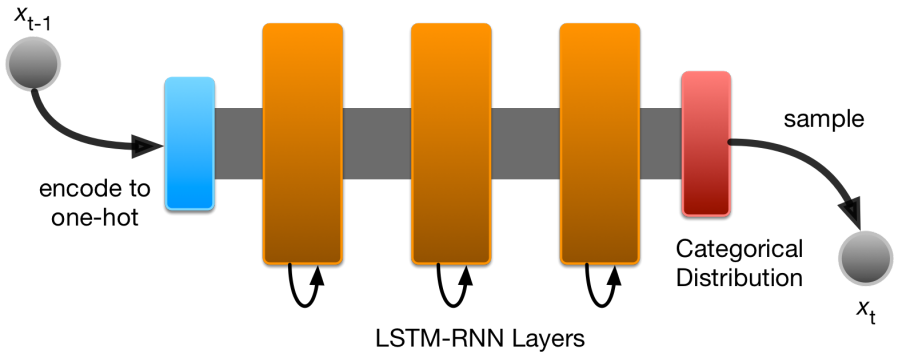
- Remember that most RNNs (and most deep learning models) end with a softmax layer.
- This layer outputs a probability distribution for a set of categorical predictions.

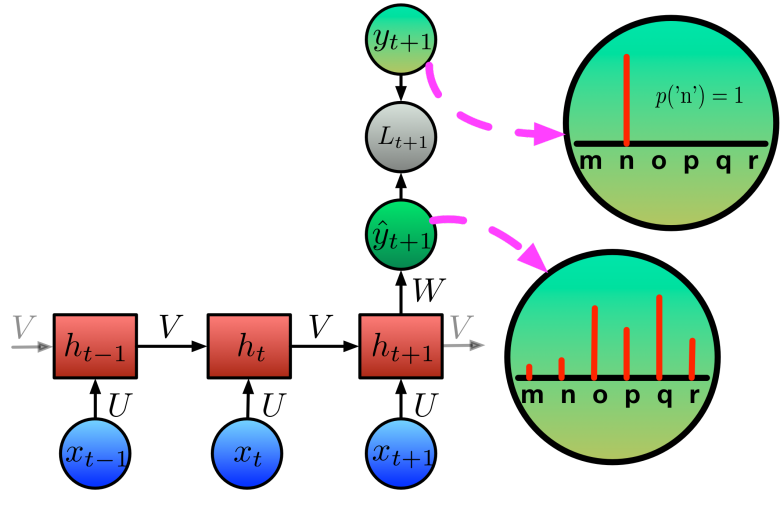




SO FAR; RNNS THAT MODEL CATEGORICAL DATA

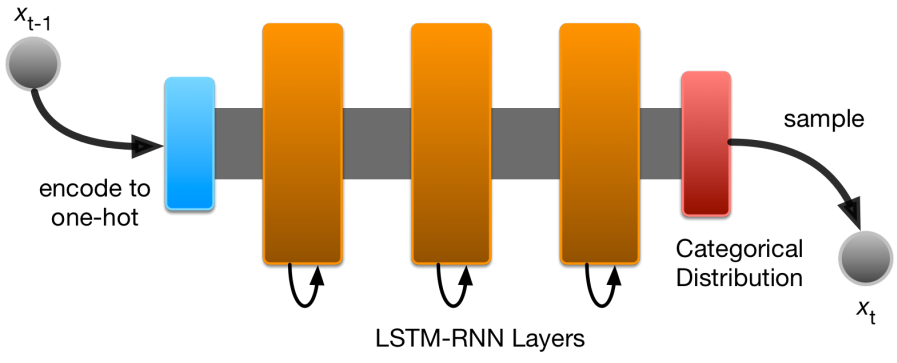
- Remember that most RNNs (and most deep learning models) end with a softmax layer.
- This layer outputs a probability distribution for a set of categorical predictions.
- E.g.:

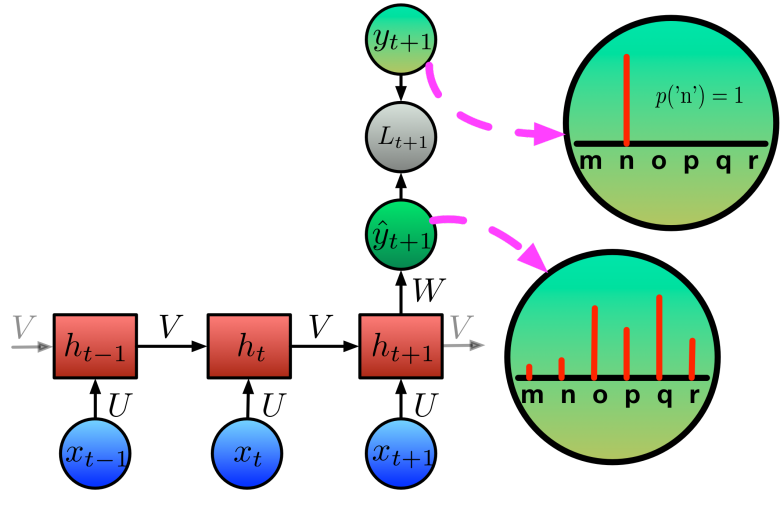




SO FAR; RNNS THAT MODEL CATEGORICAL DATA

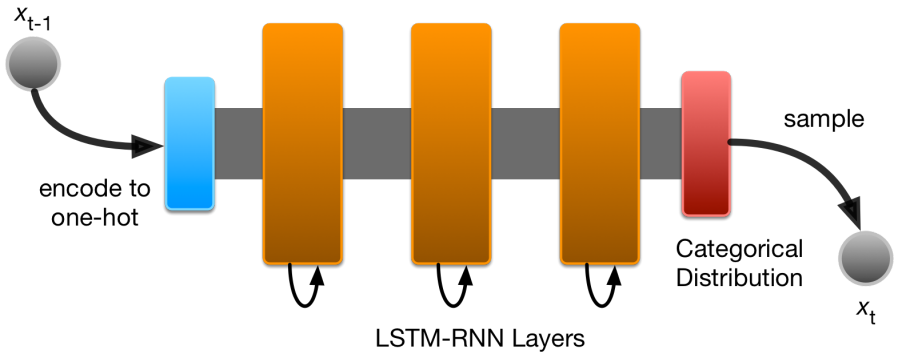
- Remember that most RNNs (and most deep learning models) end with a softmax layer.
- This layer outputs a probability distribution for a set of categorical predictions.
- E.g.:
  - image labels,

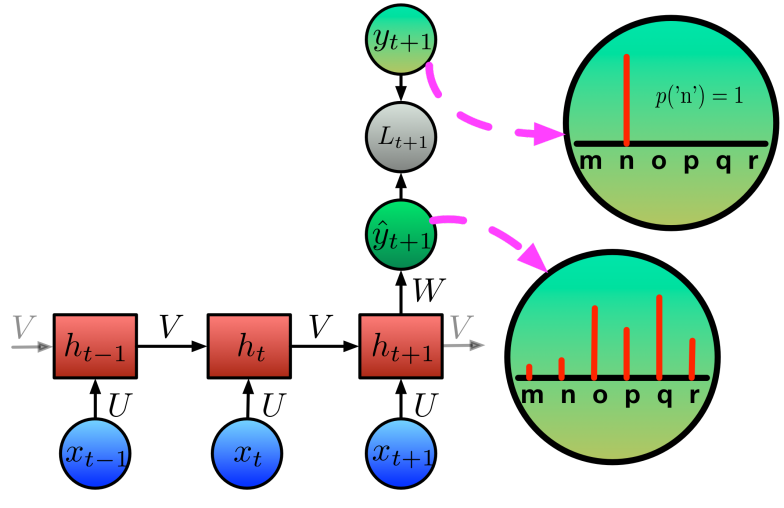




SO FAR; RNNS THAT MODEL CATEGORICAL DATA

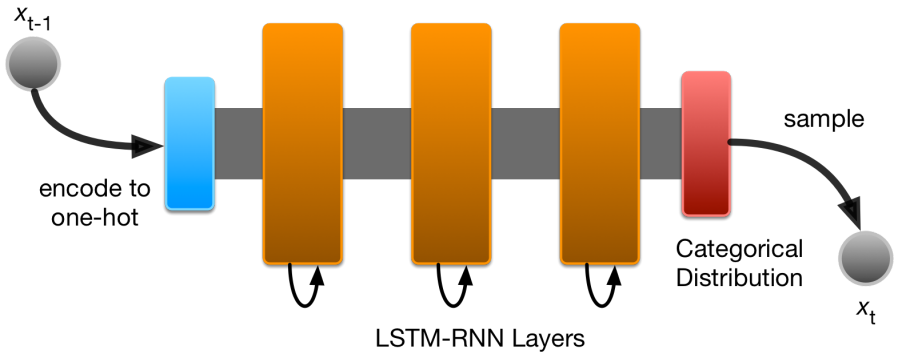
- Remember that most RNNs (and most deep learning models) end with a softmax layer.
- This layer outputs a probability distribution for a set of categorical predictions.
- E.g.:
  - image labels,
  - letters, words,

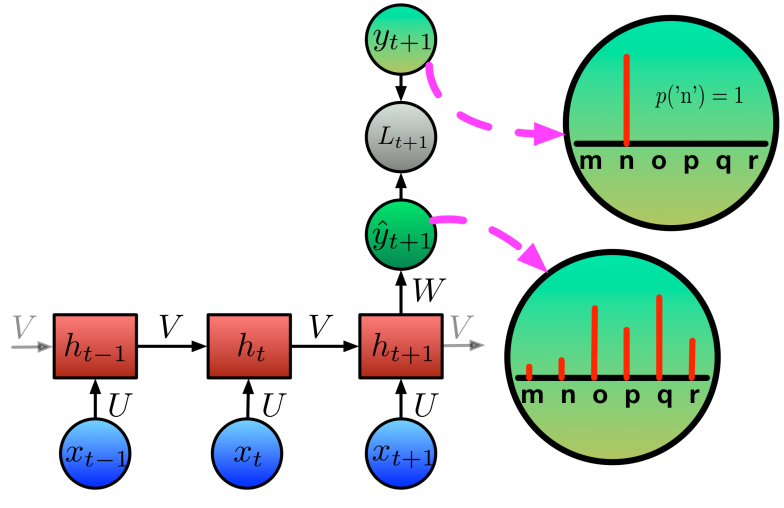




SO FAR; RNNS THAT MODEL CATEGORICAL DATA

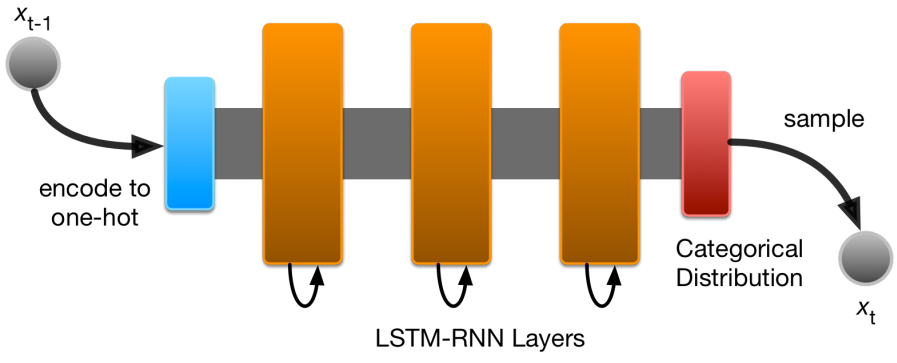
- Remember that most RNNs (and most deep learning models) end with a softmax layer.
- This layer outputs a probability distribution for a set of categorical predictions.
- E.g.:
  - image labels,
  - letters, words,
  - musical notes,

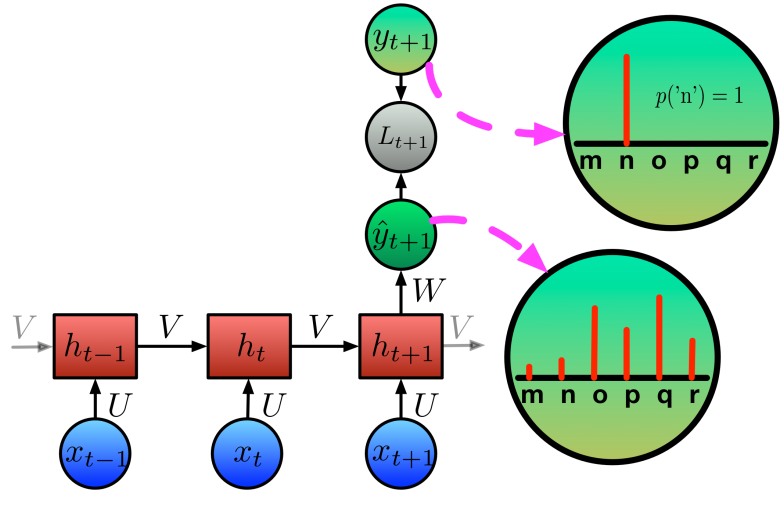




SO FAR; RNNS THAT MODEL CATEGORICAL DATA

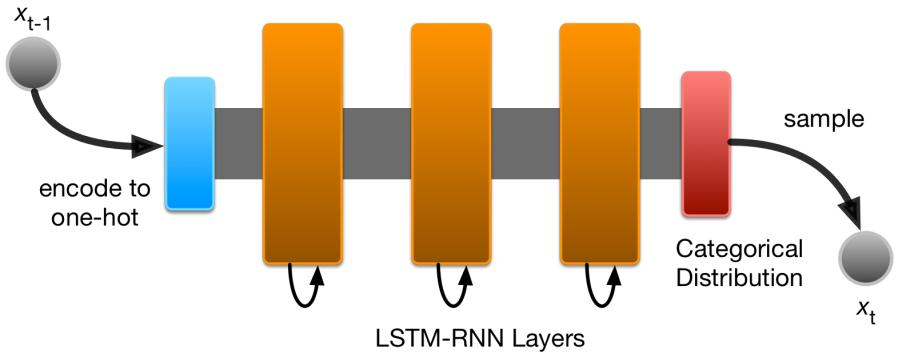
- Remember that most RNNs (and most deep learning models) end with a softmax layer.
- This layer outputs a probability distribution for a set of categorical predictions.
- E.g.:
  - image labels,
  - letters, words,
  - musical notes,
  - robot commands,

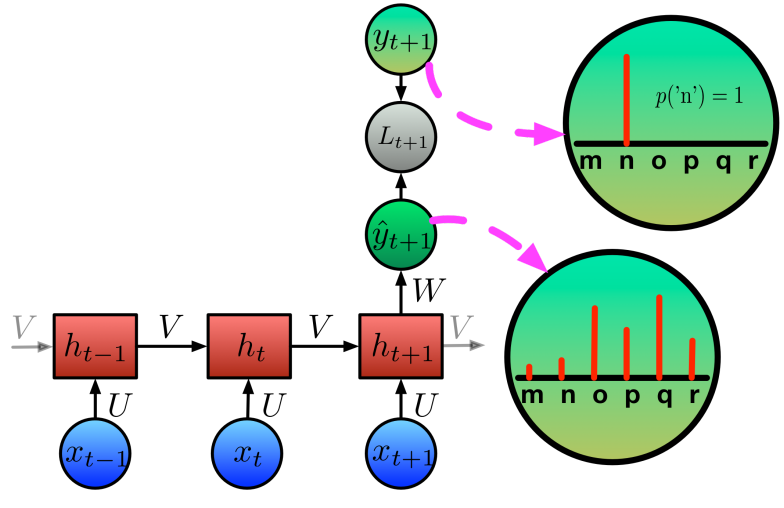




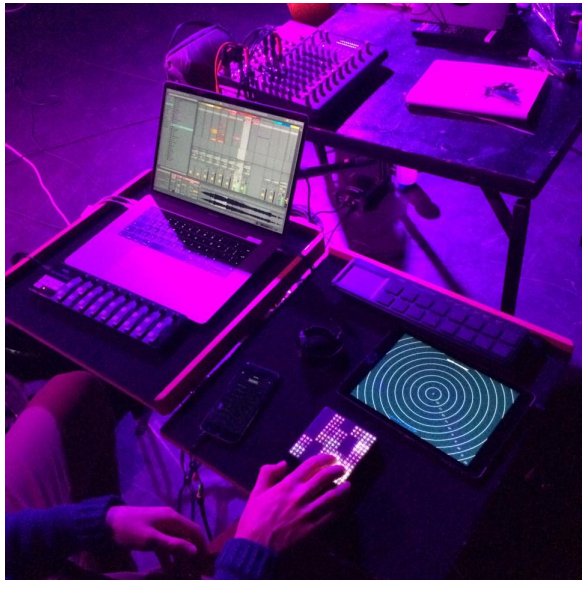
SO FAR; RNNS THAT MODEL CATEGORICAL DATA

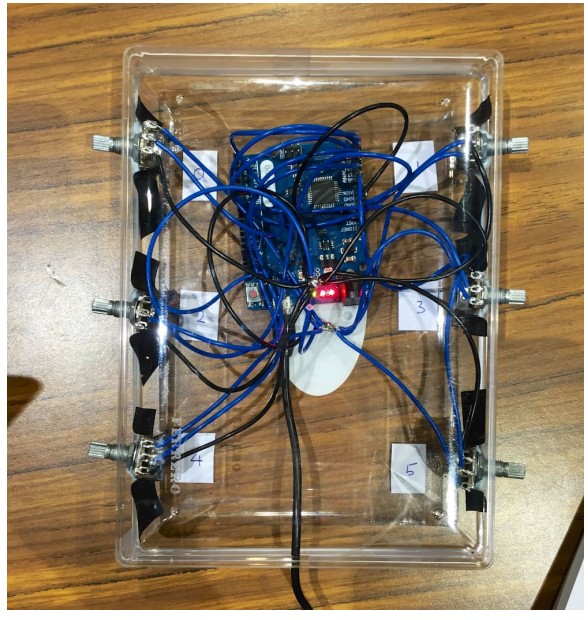
- Remember that most RNNs (and most deep learning models) end with a softmax layer.
- This layer outputs a probability distribution for a set of categorical predictions.
- E.g.:
  - image labels,
  - letters, words,
  - musical notes,
  - robot commands,
  - moves in chess.





EXPRESSIVE DATA IS OFTEN CONTINUOUS

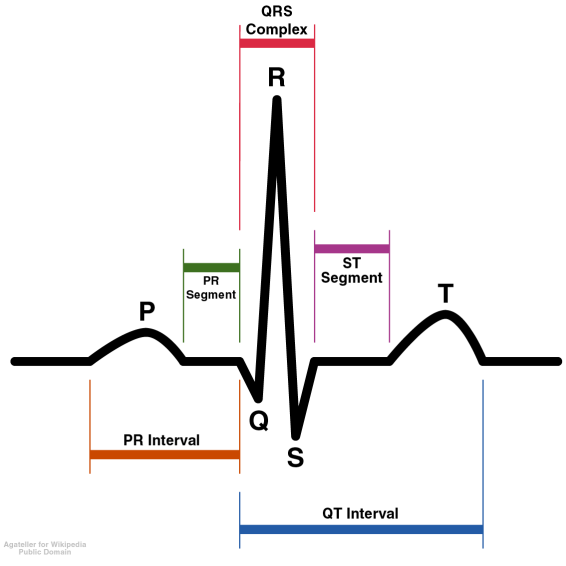




The Snyderphonics Birl Electronic Wind Instrument (Sneak Preview)

Jeff Snyder of **Snyderphonics**, developer of the **Manta** controller, shared this video preview of the upcoming **Birl** electronic wind instrument.

SO ARE BIO-SIGNALS



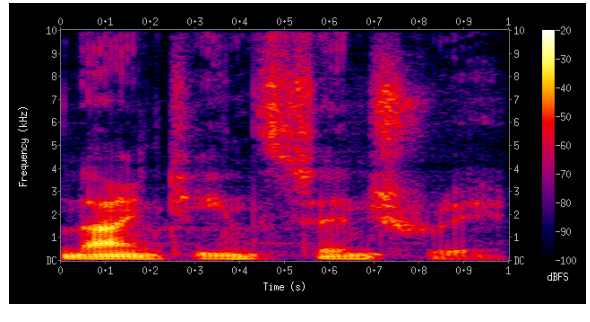
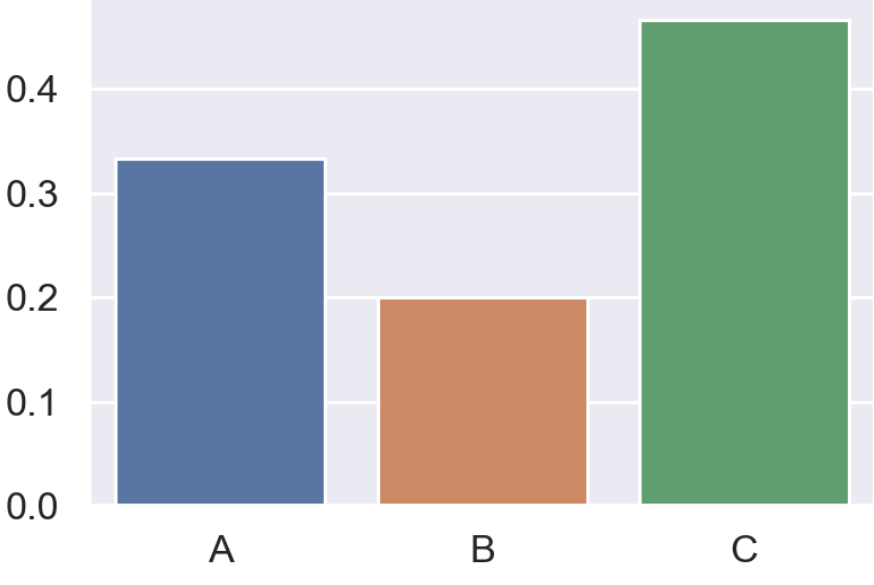
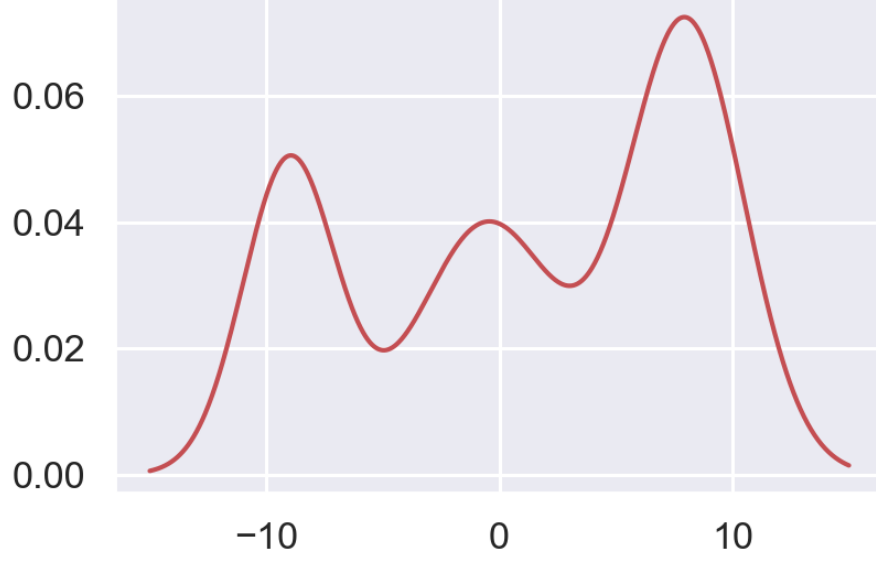


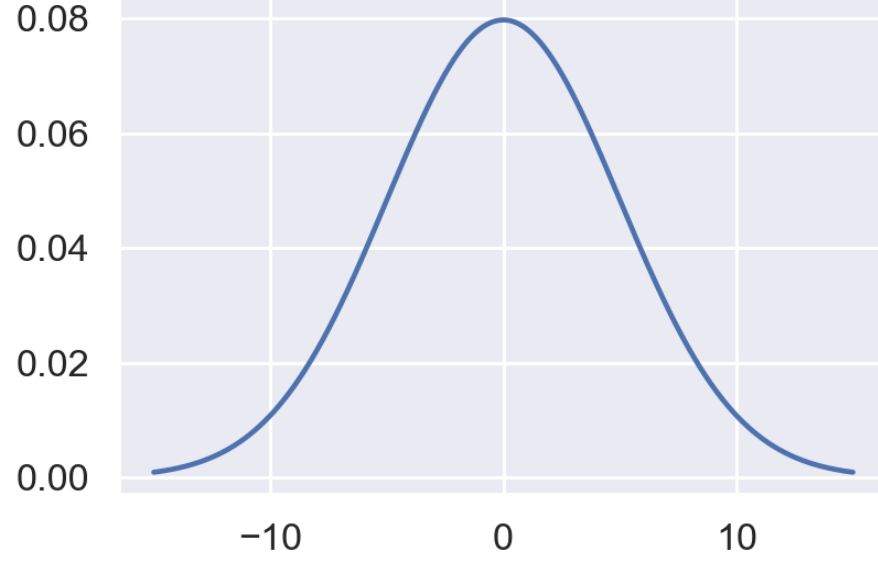
Image Credit: Wikimedia

CATEGORICAL VS. CONTINUOUS MODELS



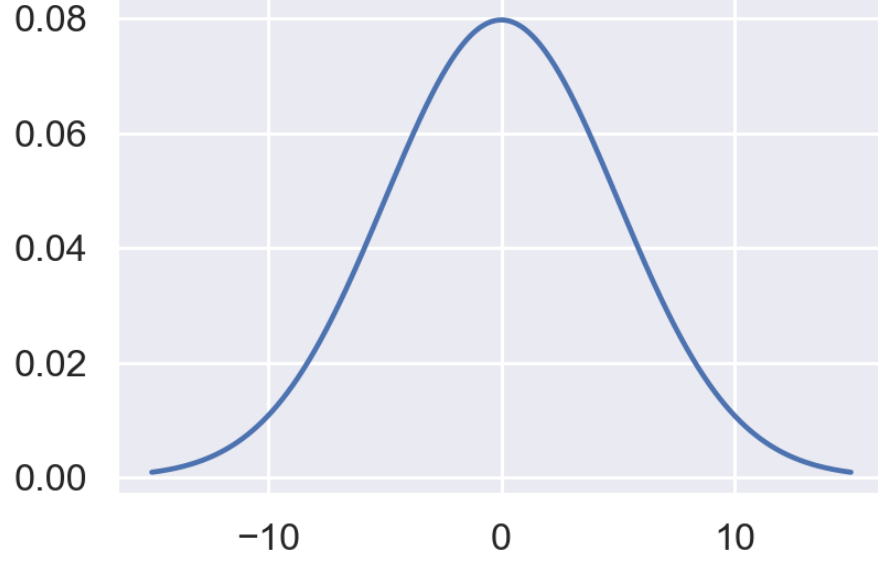


NORMAL (GAUSSIAN) DISTRIBUTION



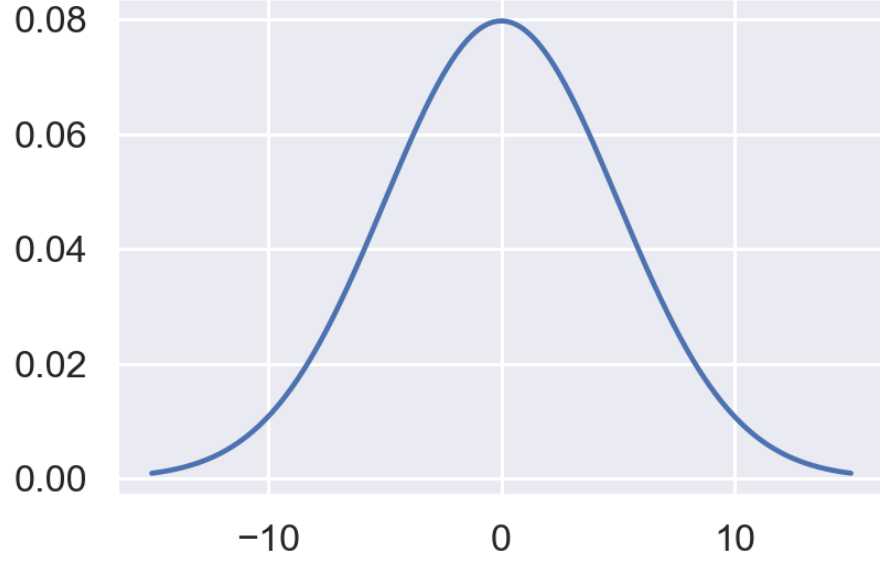
NORMAL (GAUSSIAN) DISTRIBUTION

- “Standard” probability distribution



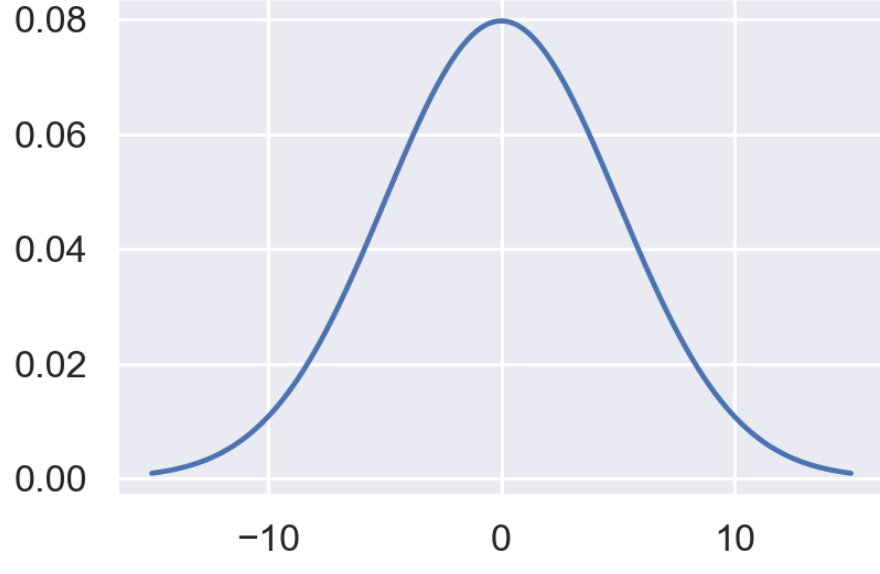
NORMAL (GAUSSIAN) DISTRIBUTION

- “Standard” probability distribution
- Has two parameters:



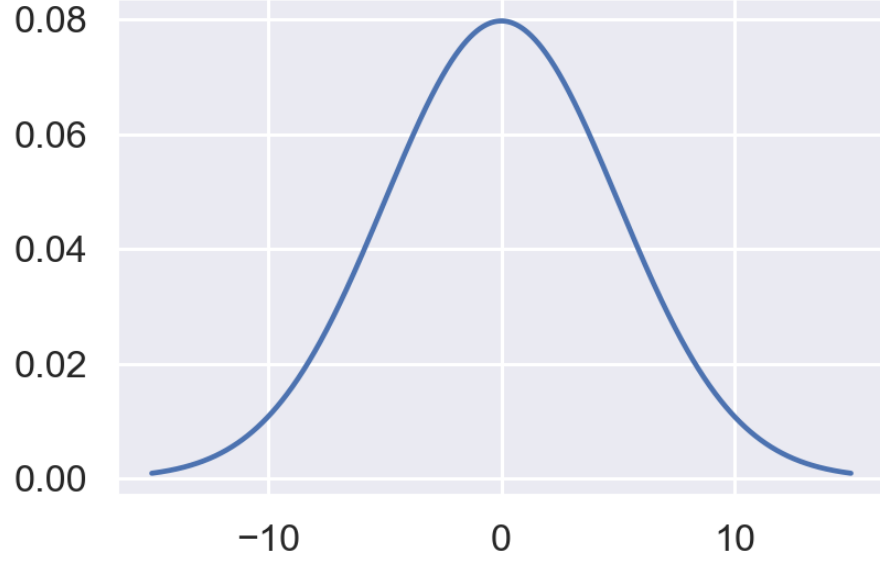
NORMAL (GAUSSIAN) DISTRIBUTION

- “Standard” probability distribution
- Has two parameters:
  - mean (μ) and



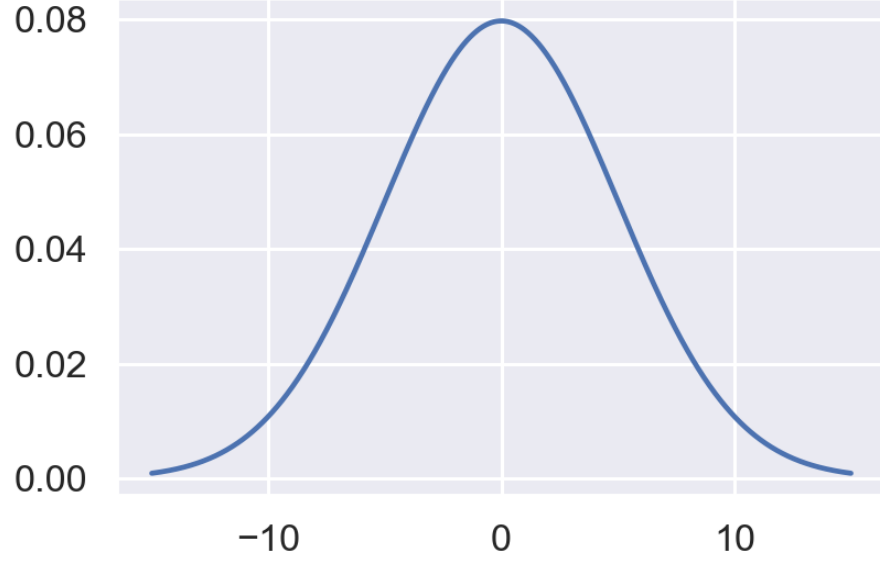
NORMAL (GAUSSIAN) DISTRIBUTION

- “Standard” probability distribution
- Has two parameters:
  - mean (μ) and
  - standard deviation (σ)



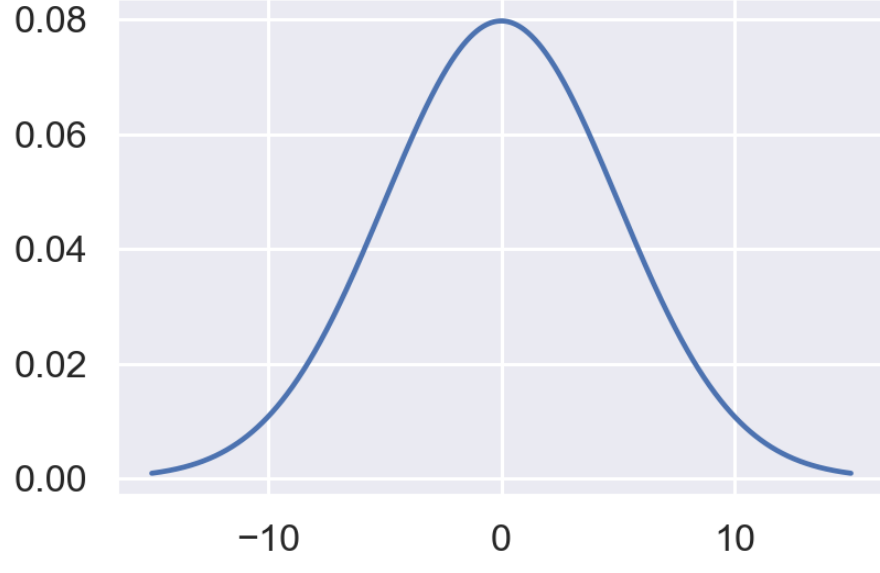
NORMAL (GAUSSIAN) DISTRIBUTION

- “Standard” probability distribution
- Has two parameters:
  - mean (μ) and
  - standard deviation (σ)
- Probability Density Function:



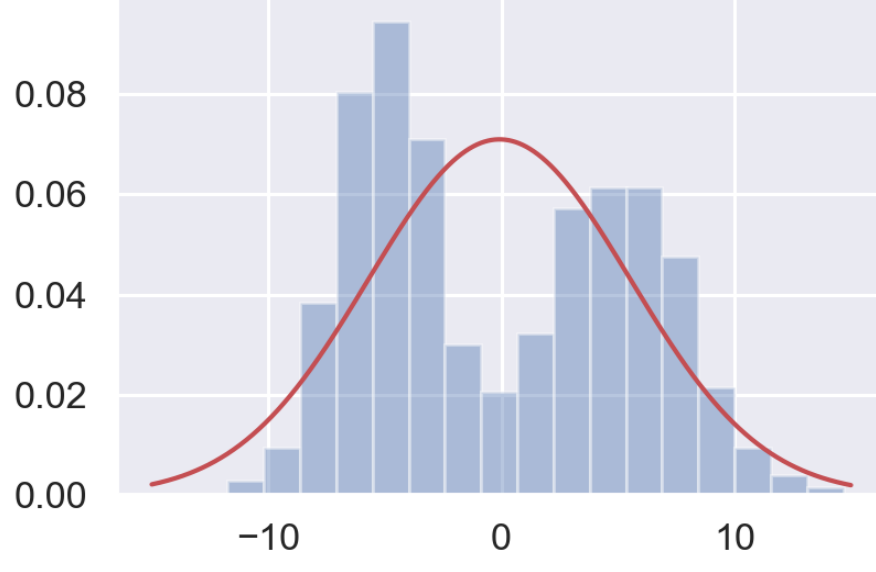
NORMAL (GAUSSIAN) DISTRIBUTION

- “Standard” probability distribution
- Has two parameters:
  - mean (μ) and
  - standard deviation (σ)
- Probability Density Function:
  - $$N(x \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$



PROBLEM: NORMAL DISTRIBUTION MIGHT NOT FIT DATA

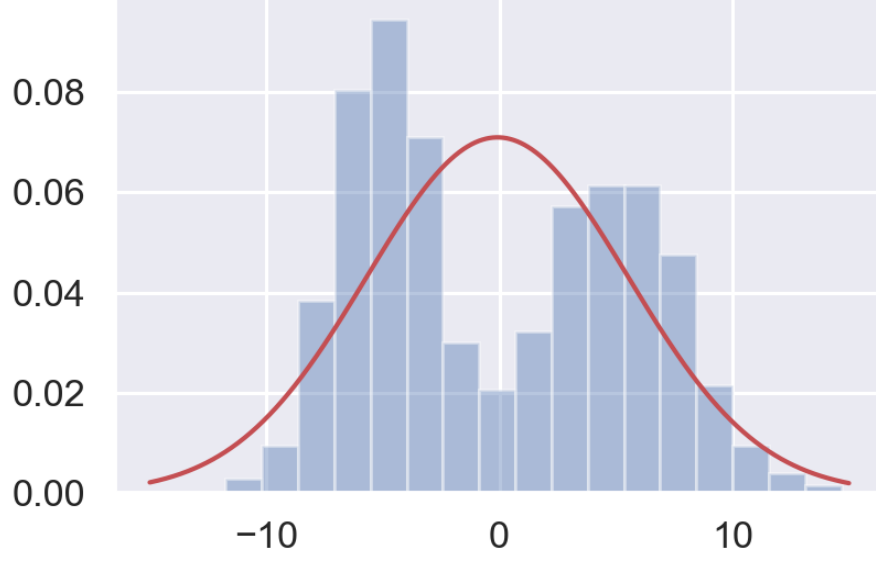
What if the data is complicated?



PROBLEM: NORMAL DISTRIBUTION MIGHT NOT FIT DATA

What if the data is complicated?

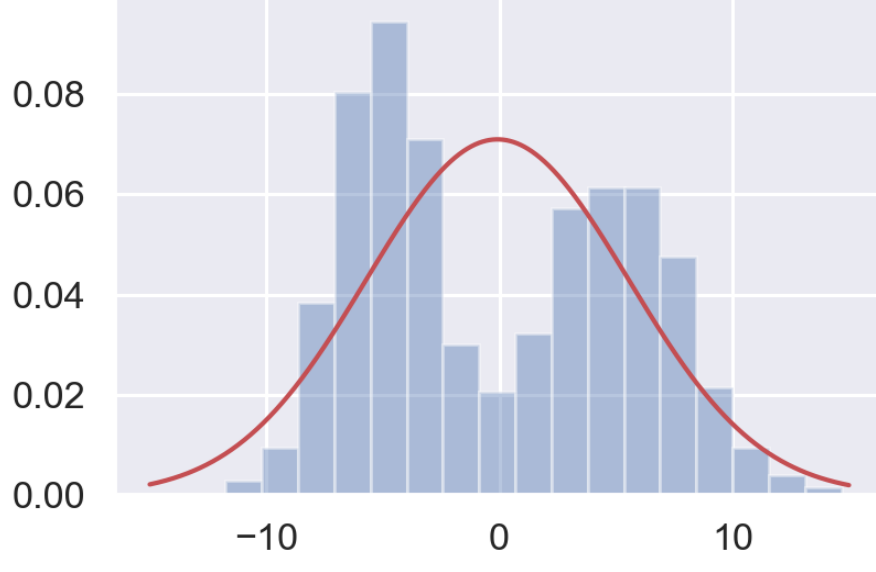
- It's easy to "fit" a normal model to any data.



PROBLEM: NORMAL DISTRIBUTION MIGHT NOT FIT DATA

What if the data is complicated?

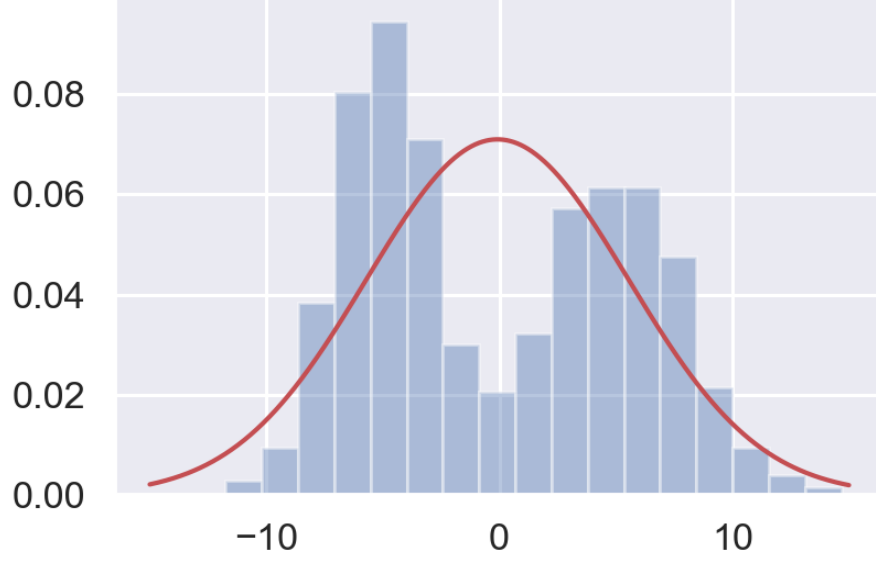
- It's easy to "fit" a normal model to any data.
  - Just calculate μ and σ



PROBLEM: NORMAL DISTRIBUTION MIGHT NOT FIT DATA

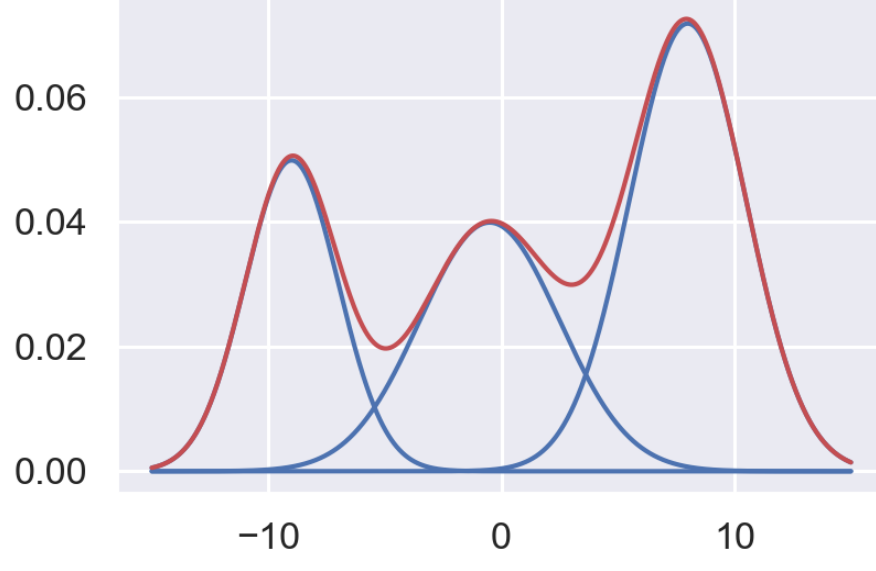
What if the data is complicated?

- It's easy to "fit" a normal model to any data.
  - Just calculate μ and σ
- But this might not fit the data well.



MIXTURE OF NORMALS

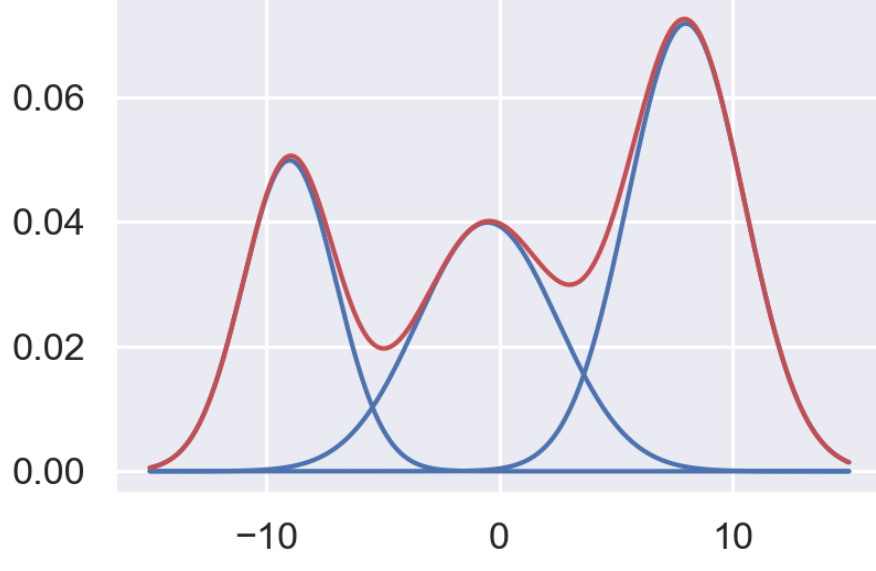
Three groups of parameters:



MIXTURE OF NORMALS

Three groups of parameters:

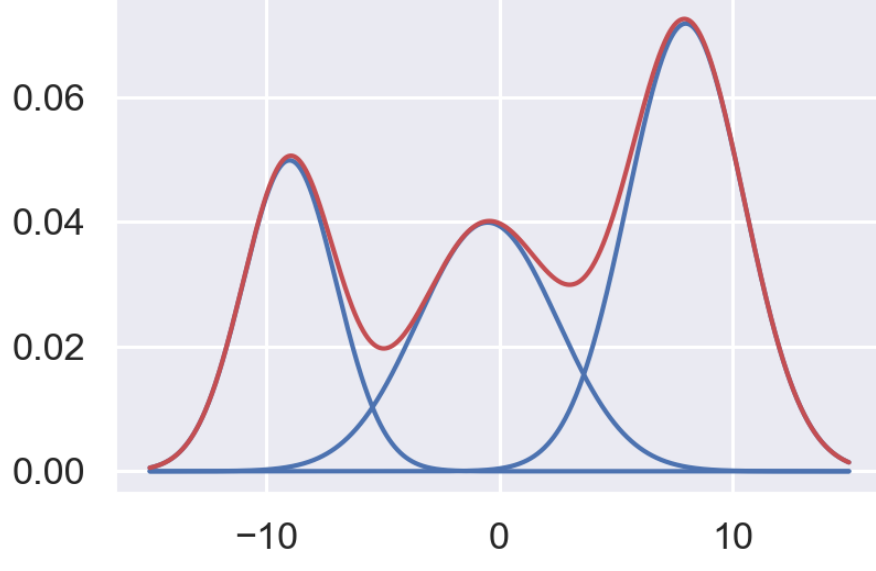
- means (μ): location of each component



MIXTURE OF NORMALS

Three groups of parameters:

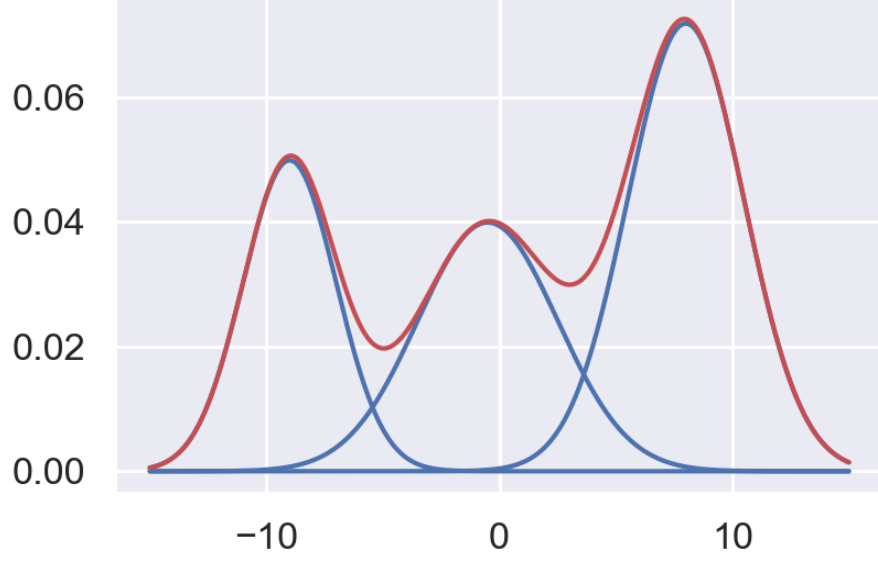
- means (μ): location of each component
- standard deviations (σ): width of each component



MIXTURE OF NORMALS

Three groups of parameters:

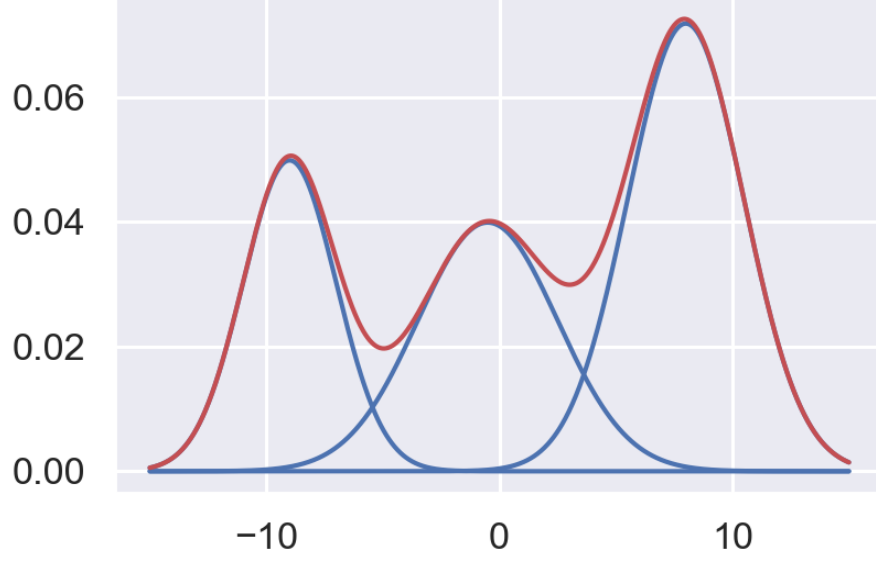
- means (μ): location of each component
- standard deviations (σ): width of each component
- Weight (π): height of each curve



MIXTURE OF NORMALS

Three groups of parameters:

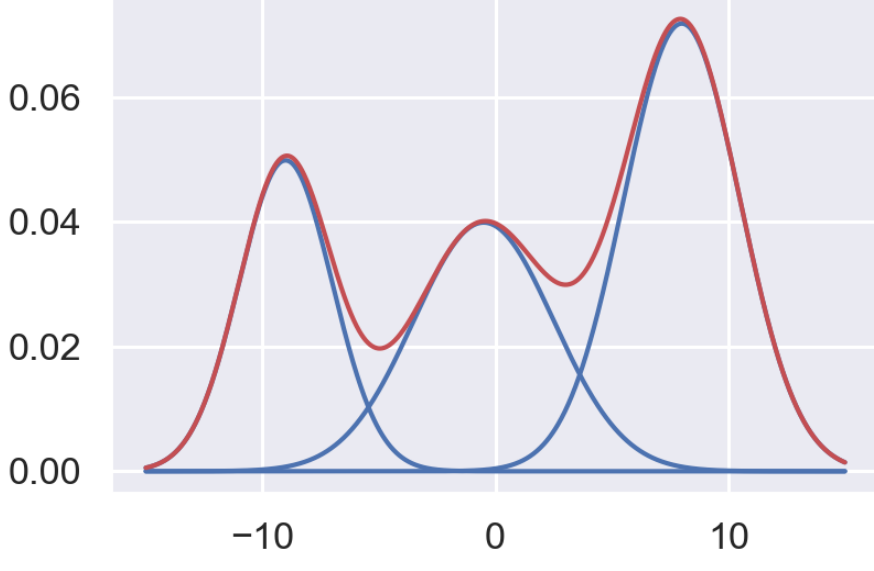
- means (μ): location of each component
- standard deviations (σ): width of each component
- Weight (π): height of each curve
- Probability Density Function:



MIXTURE OF NORMALS

Three groups of parameters:

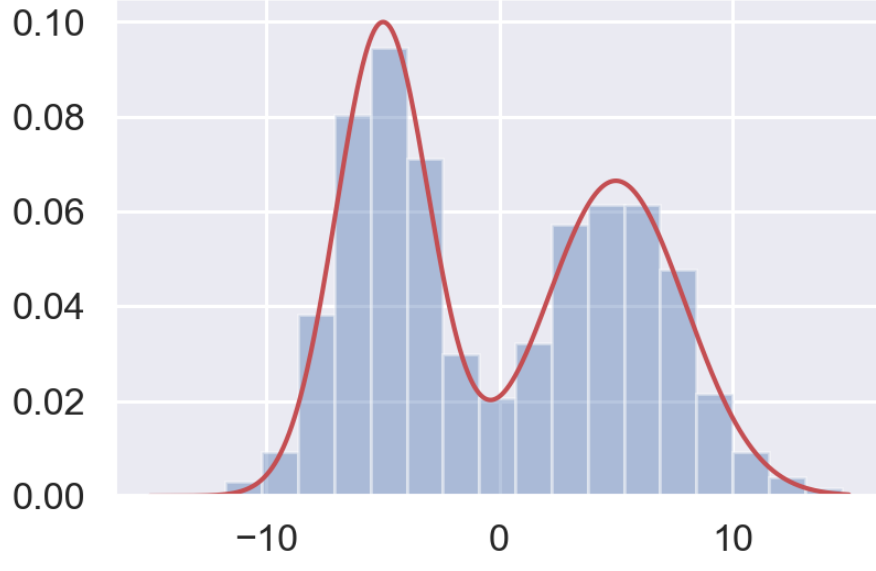
- means (μ): location of each component
- standard deviations (σ): width of each component
- Weight (π): height of each curve
- Probability Density Function:
  - $$p(x) = \sum_{i=1}^{K} \pi_i N(x \mid \mu, \sigma^2)$$



THIS SOLVES OUR PROBLEM:

Returning to our modelling problem, let's plot the PDF of a evenly-weighted mixture of the two sample normal models.

We set:



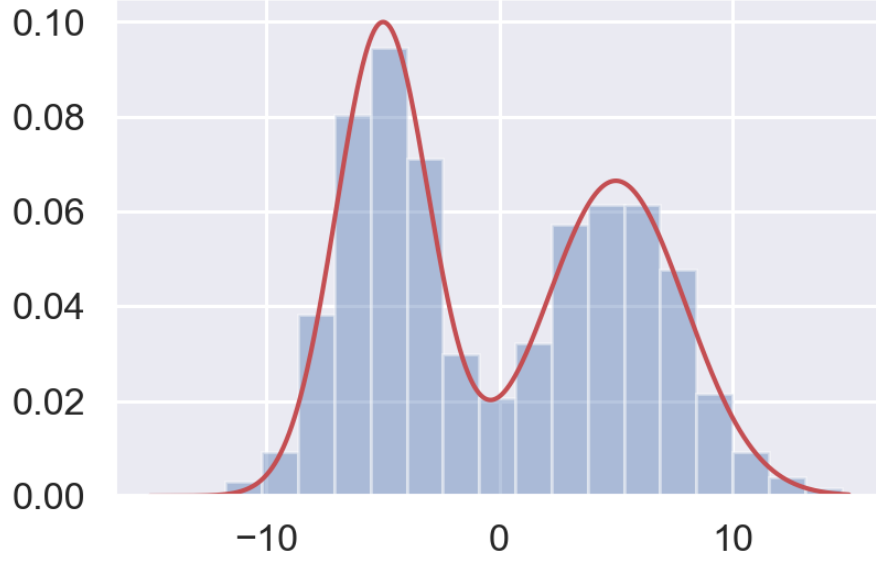
In this case, I knew the right parameters, but normally you would have to *estimate*, or *learn*, these somehow…

THIS SOLVES OUR PROBLEM:

Returning to our modelling problem, let's plot the PDF of a evenly-weighted mixture of the two sample normal models.

We set:

- K = 2



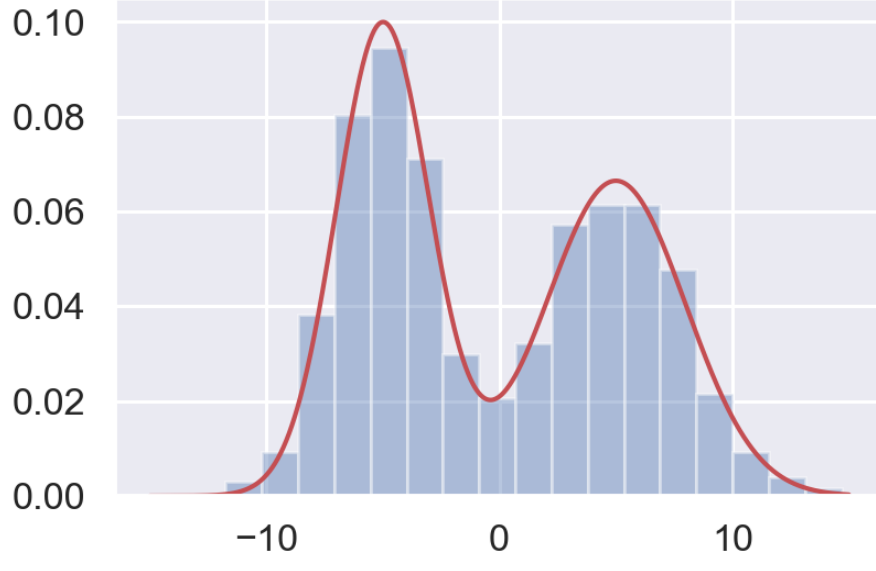
In this case, I knew the right parameters, but normally you would have to *estimate*, or *learn*, these somehow…

THIS SOLVES OUR PROBLEM:

Returning to our modelling problem, let's plot the PDF of a evenly-weighted mixture of the two sample normal models.

We set:

- $K = 2$
- $\pi = [0.5, 0.5]$



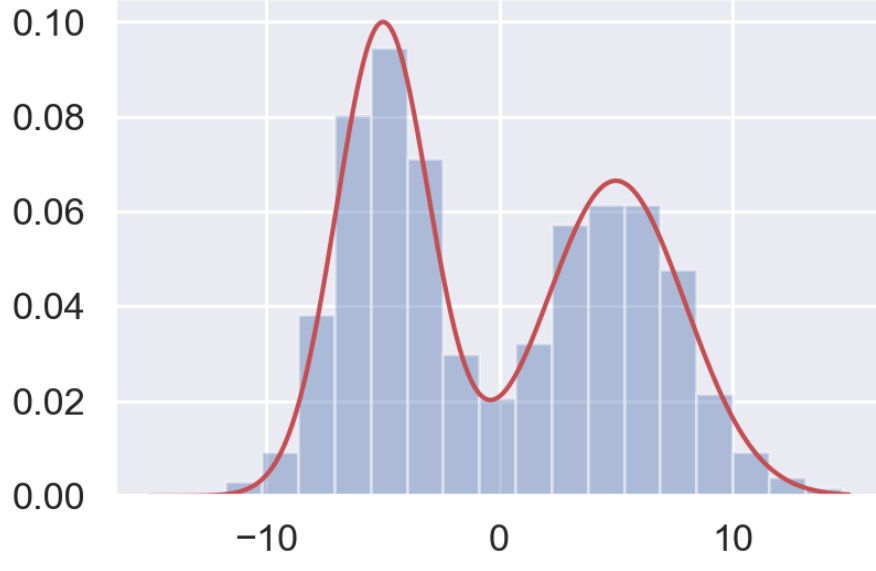
In this case, I knew the right parameters, but normally you would have to *estimate*, or *learn*, these somehow…

THIS SOLVES OUR PROBLEM:

Returning to our modelling problem, let's plot the PDF of a evenly-weighted mixture of the two sample normal models.

We set:

- $K = 2$
- $\pi = [0.5, 0.5]$
- $\mu = [-5, 5]$



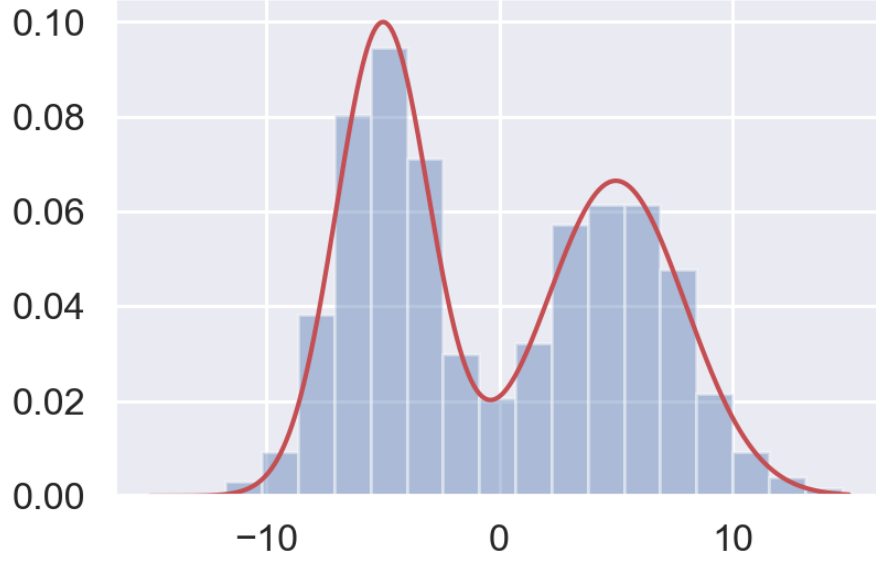
In this case, I knew the right parameters, but normally you would have to *estimate*, or *learn*, these somehow…

THIS SOLVES OUR PROBLEM:

Returning to our modelling problem, let's plot the PDF of a evenly-weighted mixture of the two sample normal models.

We set:

- $K = 2$
- $\pi = [0.5, 0.5]$
- $\mu = [-5, 5]$
- $\sigma = [2, 3]$



In this case, I knew the right parameters, but normally you would have to *estimate*, or *learn*, these somehow…

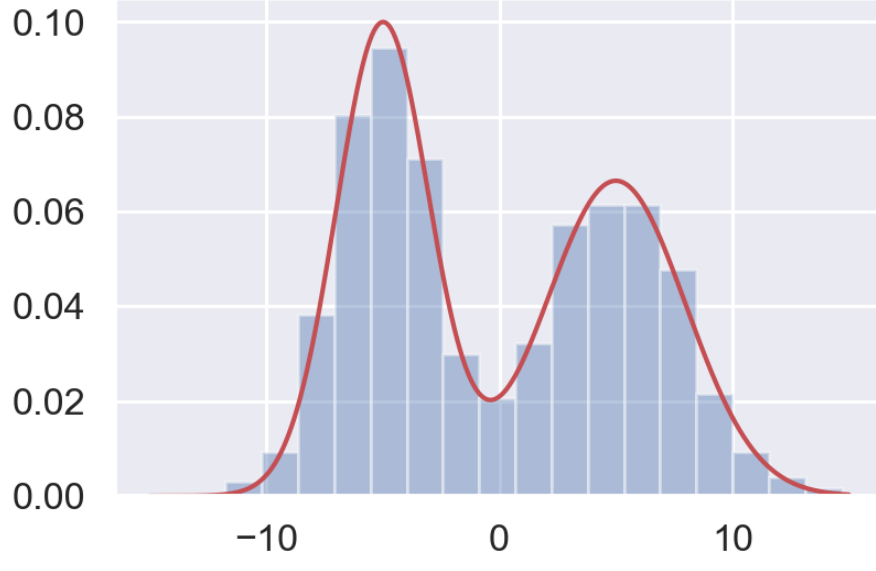
THIS SOLVES OUR PROBLEM:

Returning to our modelling problem, let's plot the PDF of a evenly-weighted mixture of the two sample normal models.

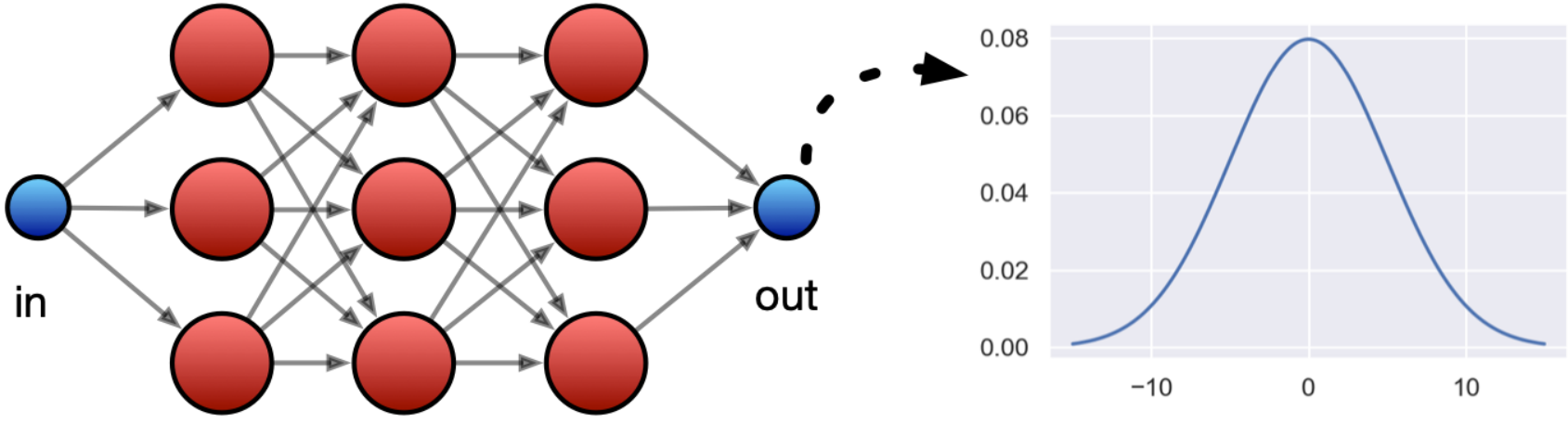
We set:

- $K = 2$
- $\pi = [0.5, 0.5]$
- $\mu = [-5, 5]$
- $\sigma = [2, 3]$
- (bold used to indicate the vector of parameters for each component)

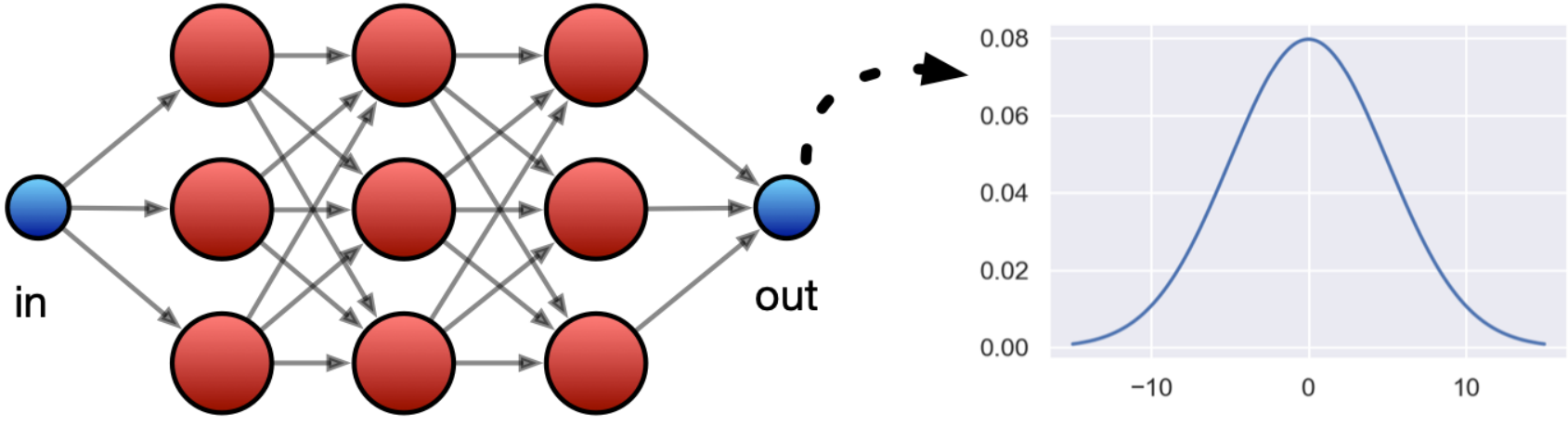
In this case, I knew the right parameters, but normally you would have to *estimate*, or *learn*, these somehow…



MIXTURE DENSITY NETWORKS

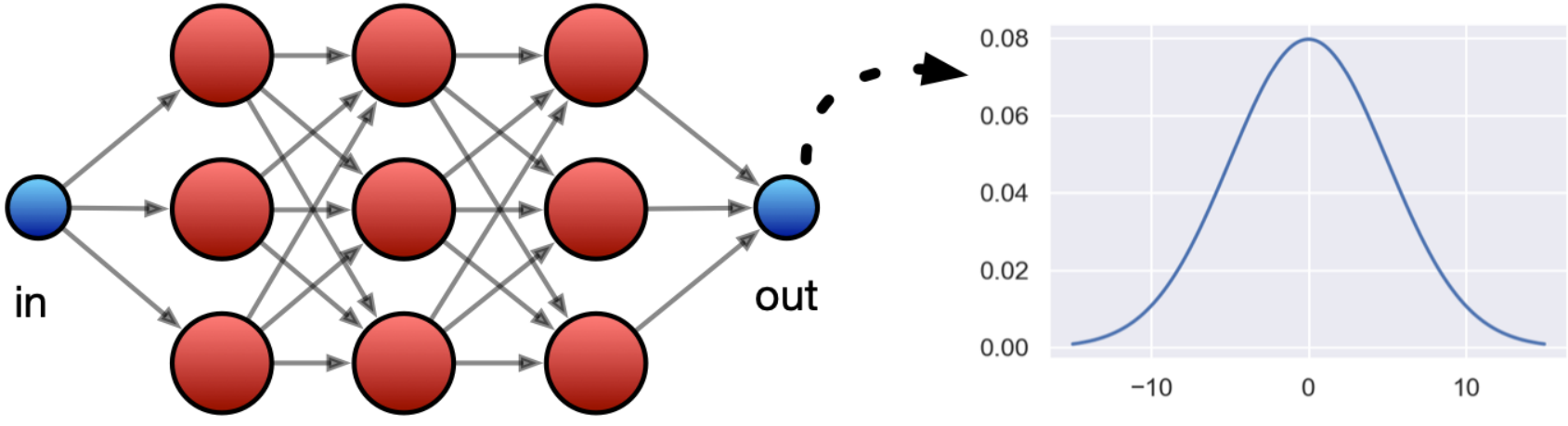


MIXTURE DENSITY NETWORKS



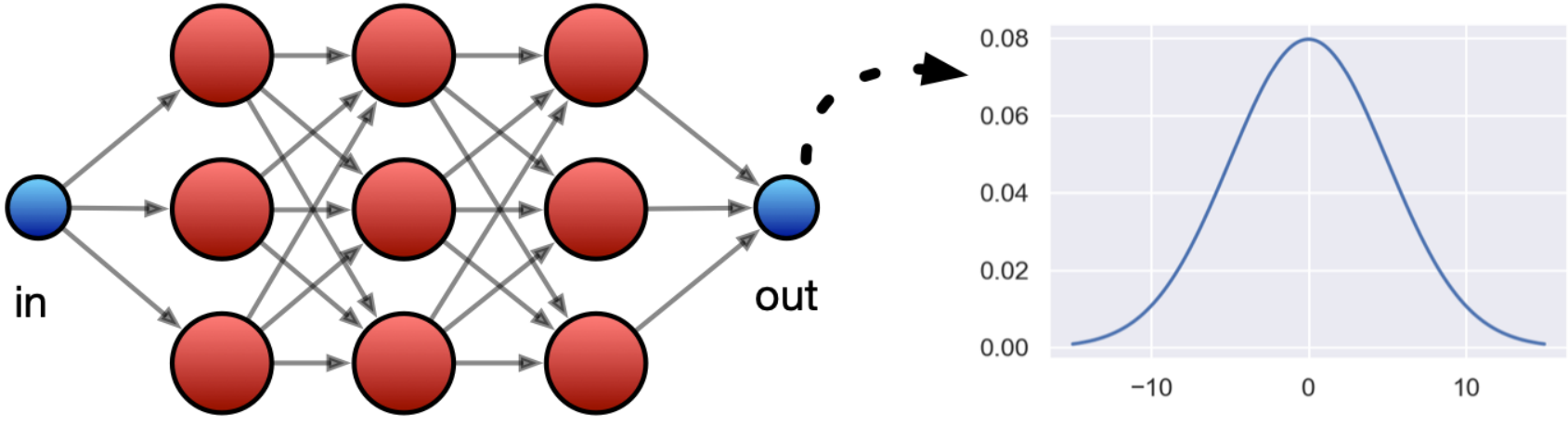
- Neural networks used to model complicated real-valued data.

MIXTURE DENSITY NETWORKS



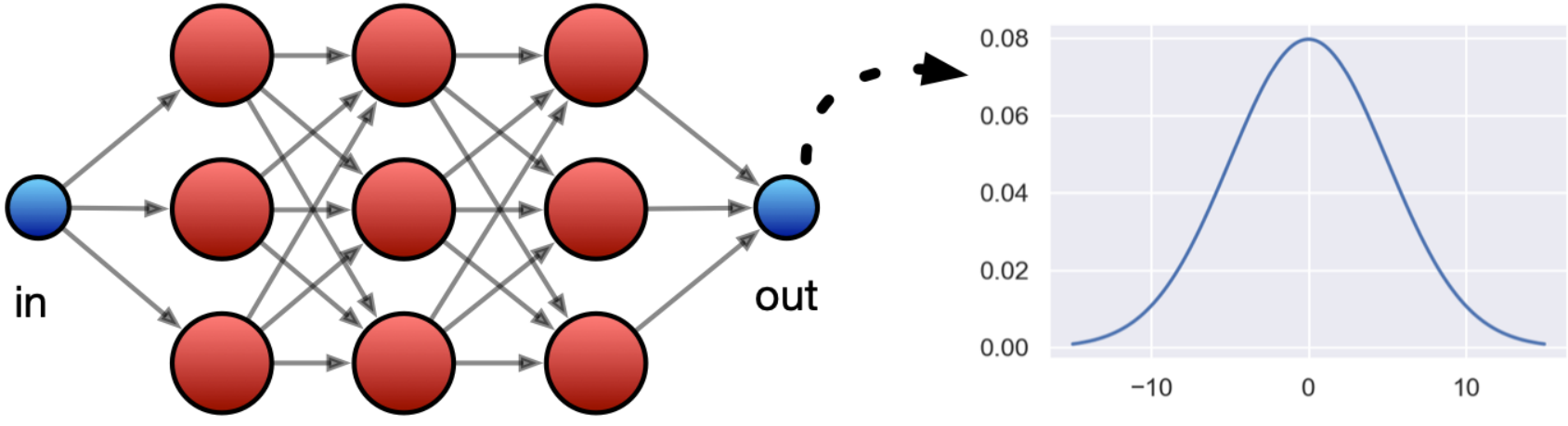
- Neural networks used to model complicated real-valued data.
  - i.e., data that might not be very “normal”

MIXTURE DENSITY NETWORKS



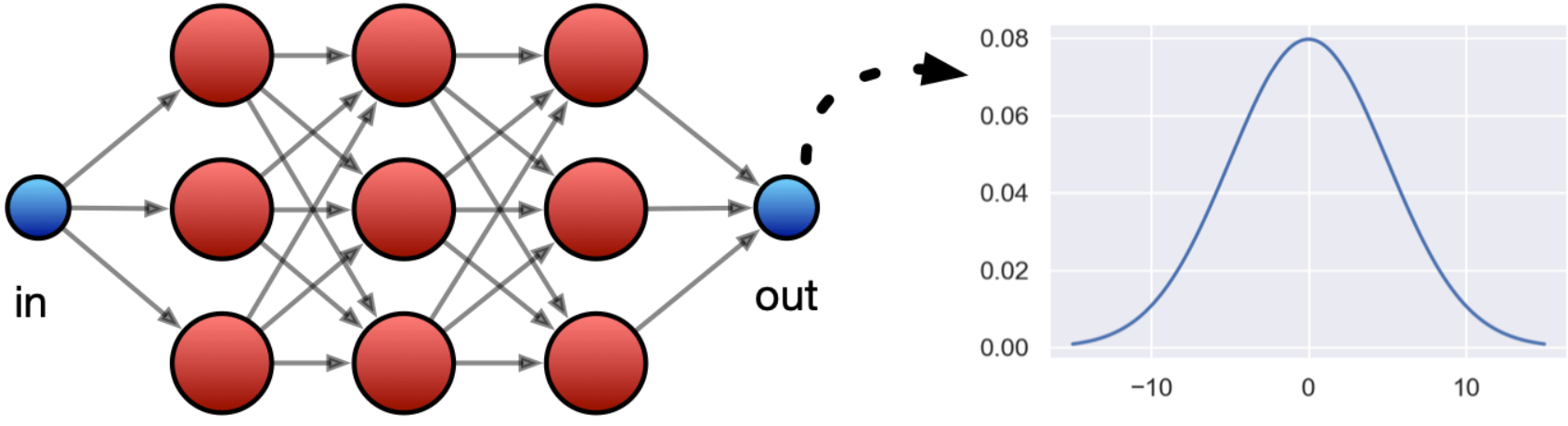
- Neural networks used to model complicated real-valued data.
  - i.e., data that might not be very “normal”
- Usual approach: use a neuron with linear activation to make predictions.

MIXTURE DENSITY NETWORKS



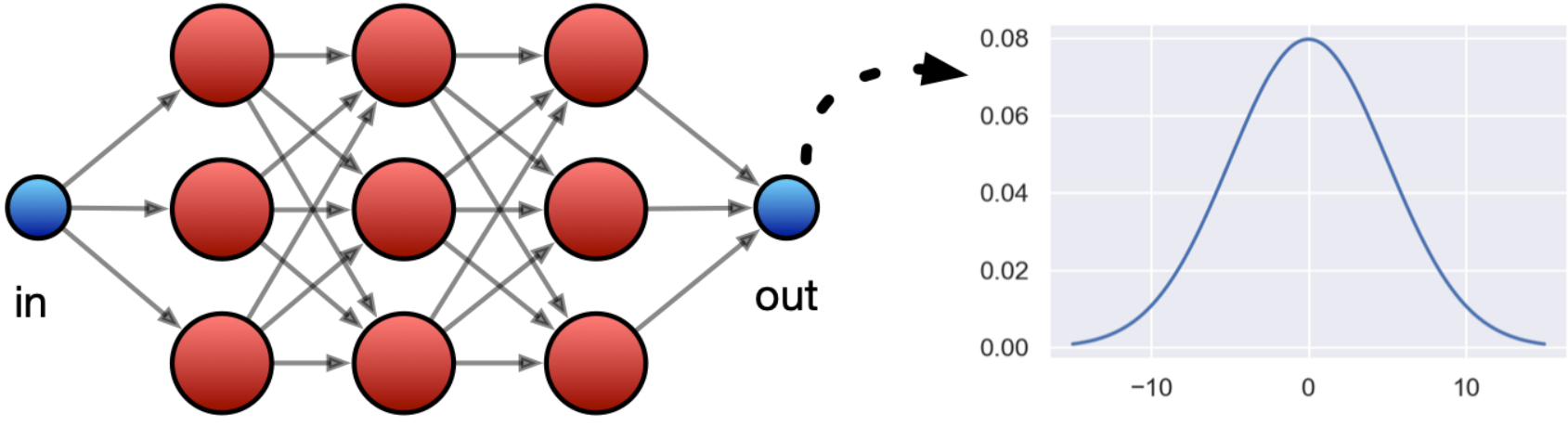
- Neural networks used to model complicated real-valued data.
  - i.e., data that might not be very “normal”
- Usual approach: use a neuron with linear activation to make predictions.
  - Training function could be MSE (mean squared error).

MIXTURE DENSITY NETWORKS



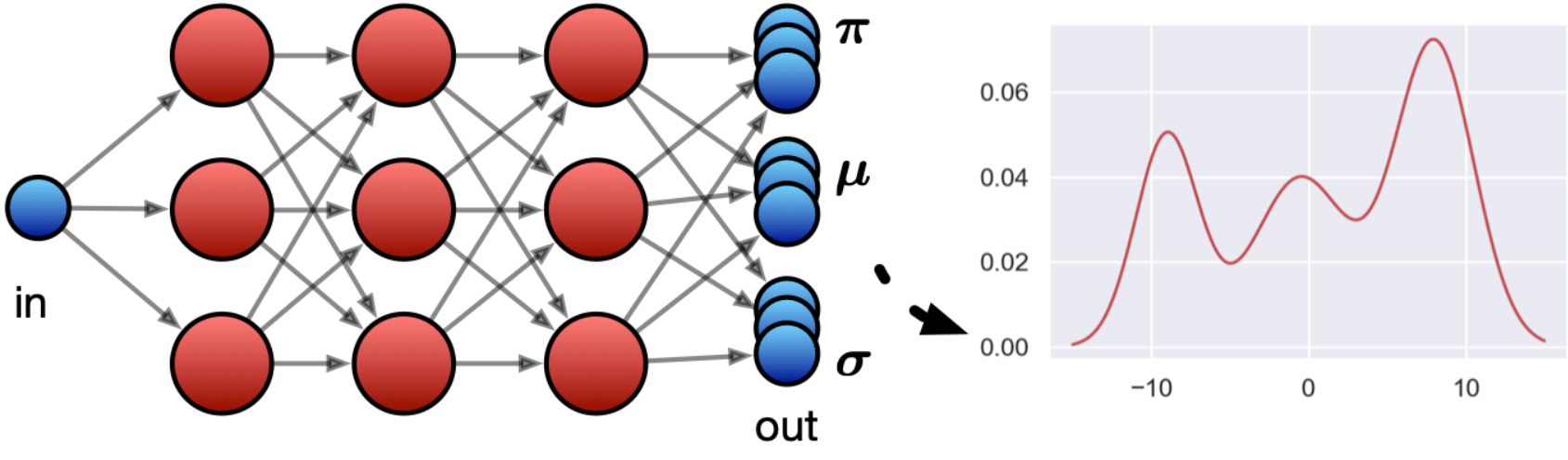
- Neural networks used to model complicated real-valued data.
  - i.e., data that might not be very “normal”
- Usual approach: use a neuron with linear activation to make predictions.
  - Training function could be MSE (mean squared error).
- Problem! This is equivalent to fitting to a single normal model! 😱

MIXTURE DENSITY NETWORKS

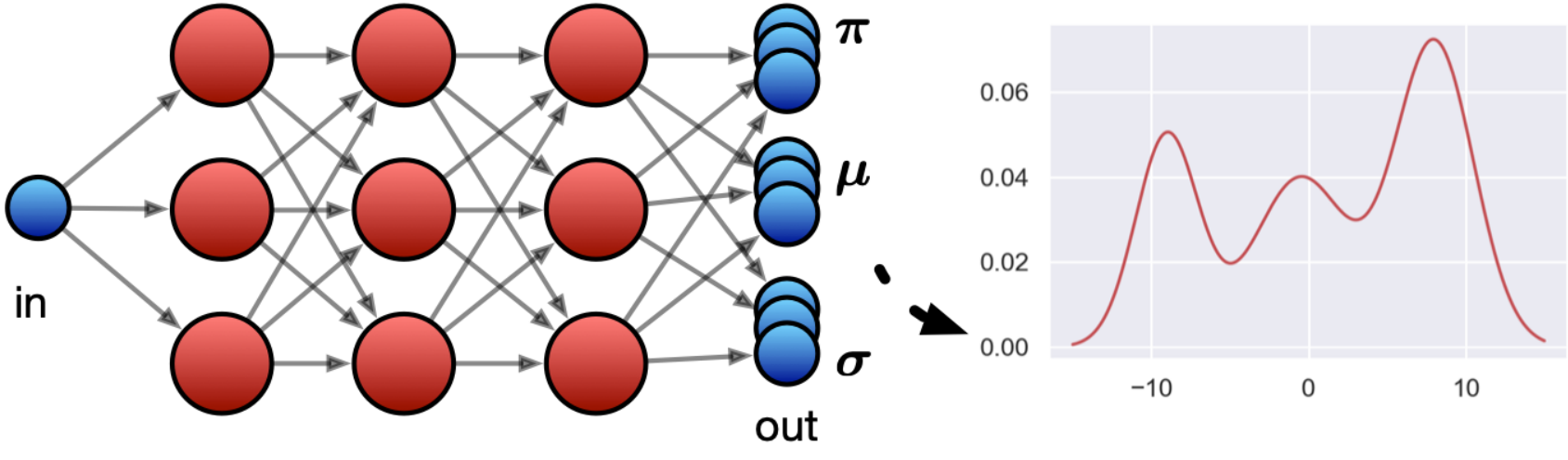


- Neural networks used to model complicated real-valued data.
  - i.e., data that might not be very “normal”
- Usual approach: use a neuron with linear activation to make predictions.
  - Training function could be MSE (mean squared error).
- Problem! This is equivalent to fitting to a single normal model! 😱
- (See Bishop, C (1994) for proof and more details)

MIXTURE DENSITY NETWORKS

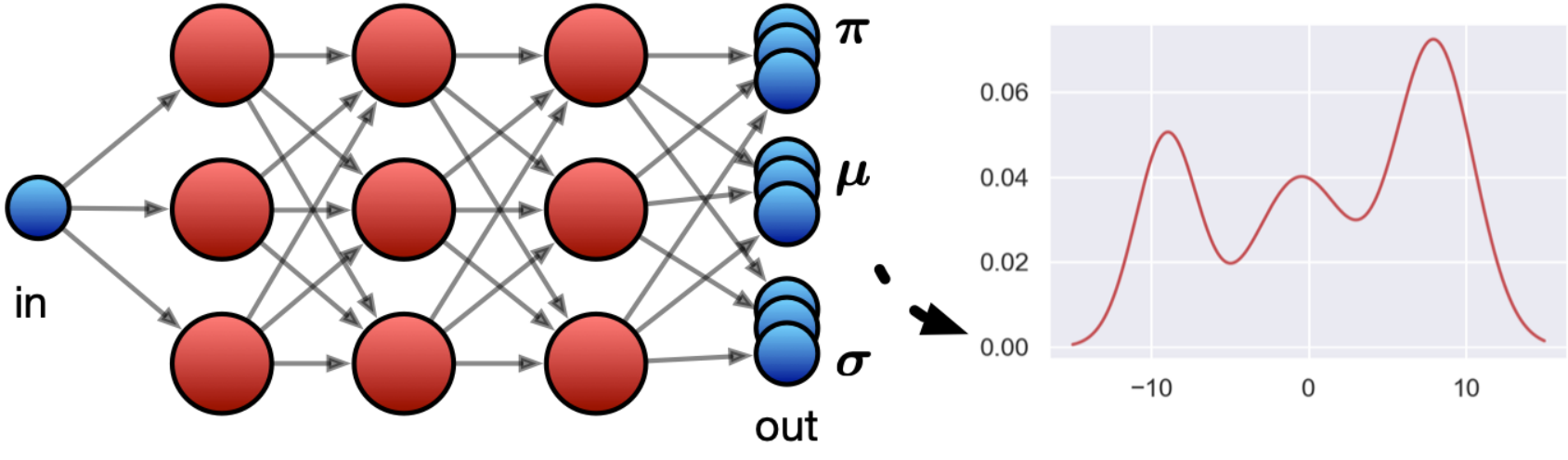


MIXTURE DENSITY NETWORKS



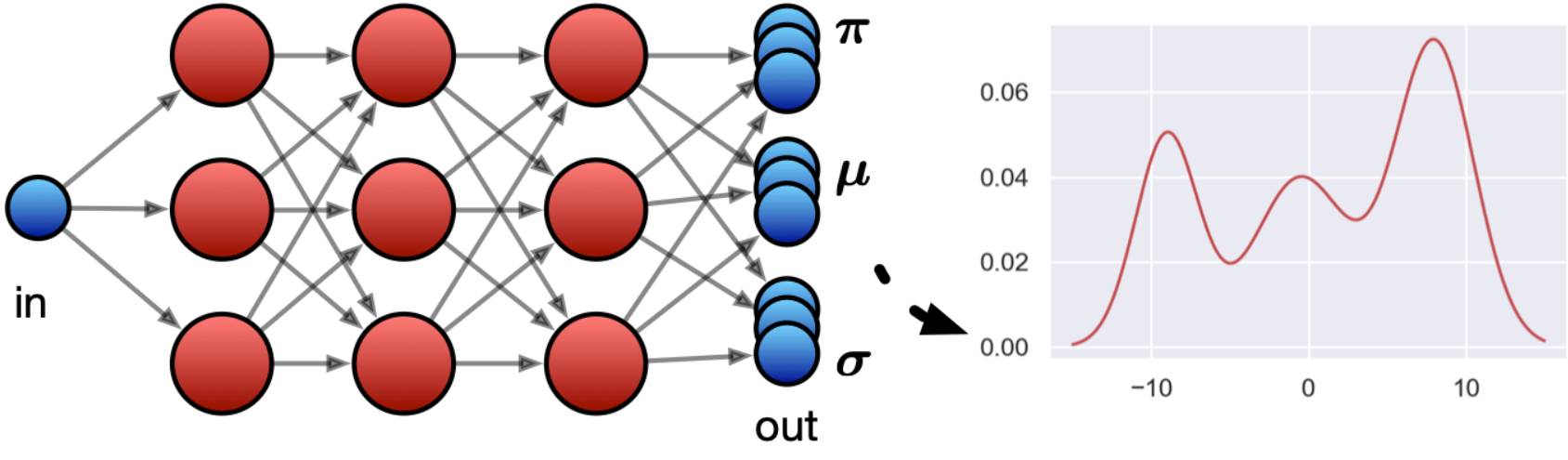
- Idea: output parameters of a mixture model instead!

MIXTURE DENSITY NETWORKS



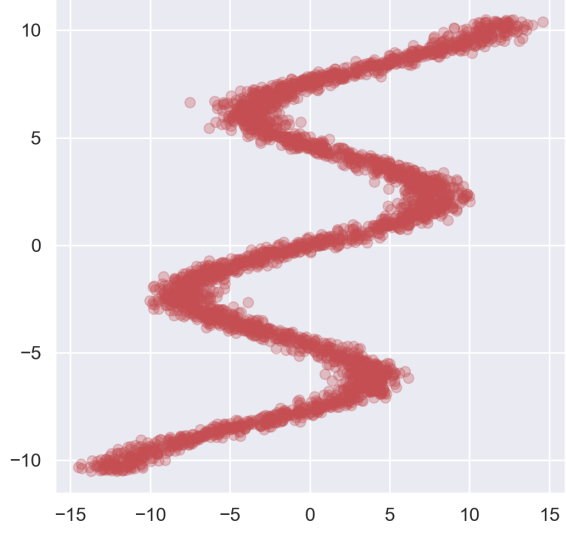
- Idea: output parameters of a mixture model instead!
- Rather than MSE for training, use the PDF of the mixture model.

MIXTURE DENSITY NETWORKS

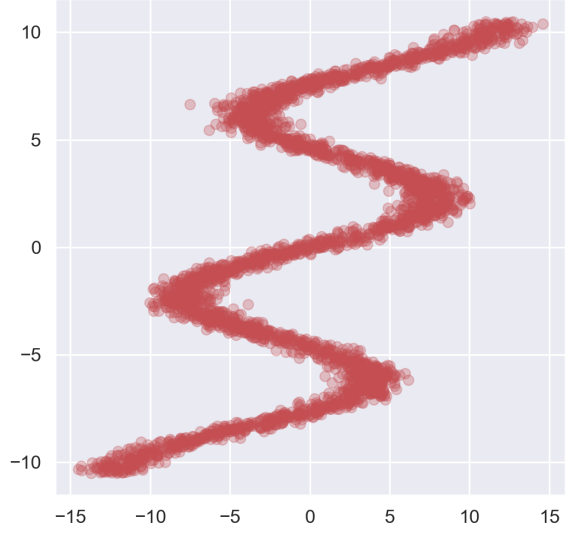


- Idea: output parameters of a mixture model instead!
- Rather than MSE for training, use the PDF of the mixture model.
- Now network can model complicated distributions! 😌

SIMPLE EXAMPLE IN KERAS

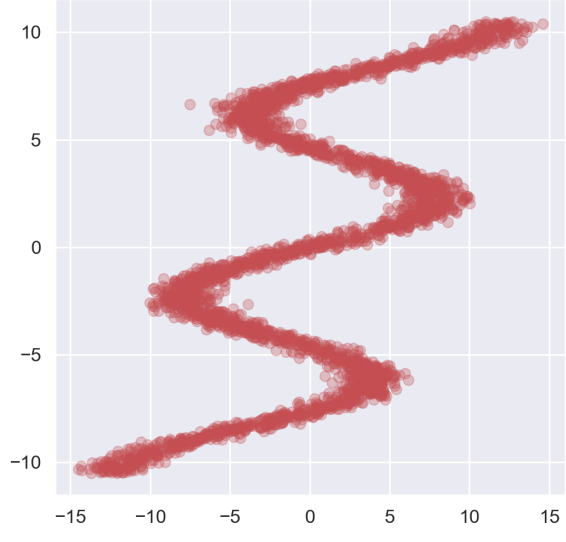


SIMPLE EXAMPLE IN KERAS



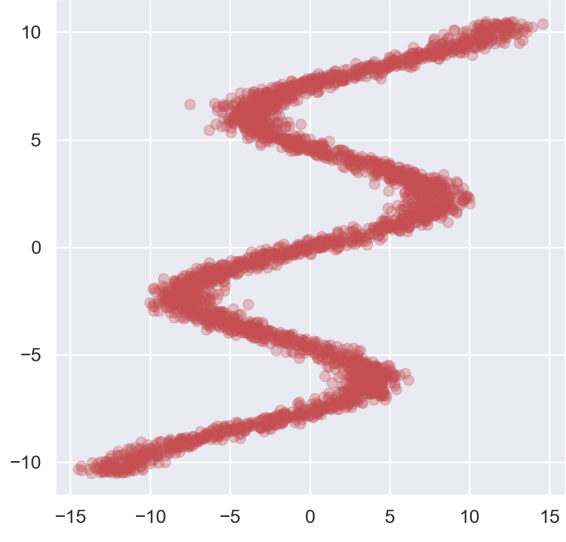
- Difficult data is not hard to find! Think about modelling an inverse sine (arcsine) function.

SIMPLE EXAMPLE IN KERAS



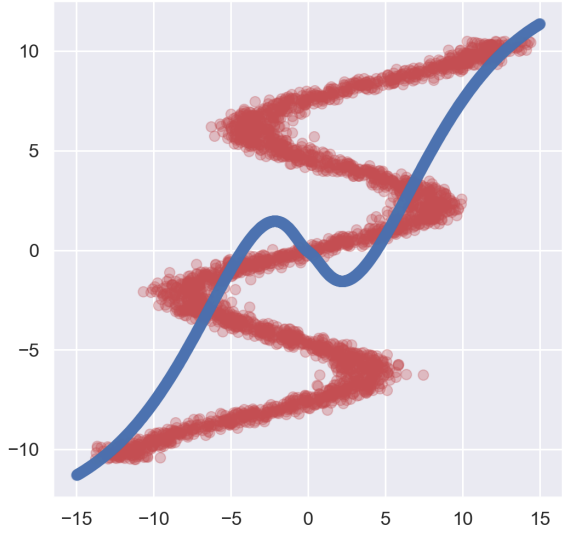
- Difficult data is not hard to find! Think about modelling an inverse sine (arcsine) function.
  - Each input value takes multiple outputs…

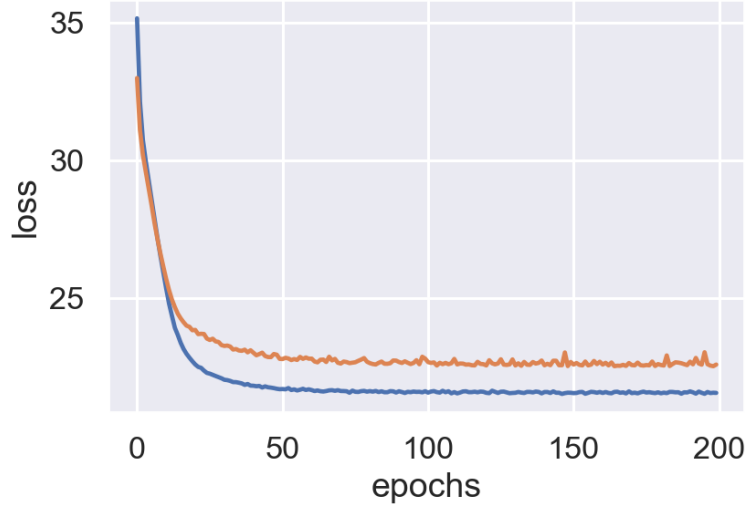
SIMPLE EXAMPLE IN KERAS



- Difficult data is not hard to find! Think about modelling an inverse sine (arcsine) function.
  - Each input value takes multiple outputs…
  - This is not going to go well for a single normal model.

FEEDFORWARD MSE NETWORK

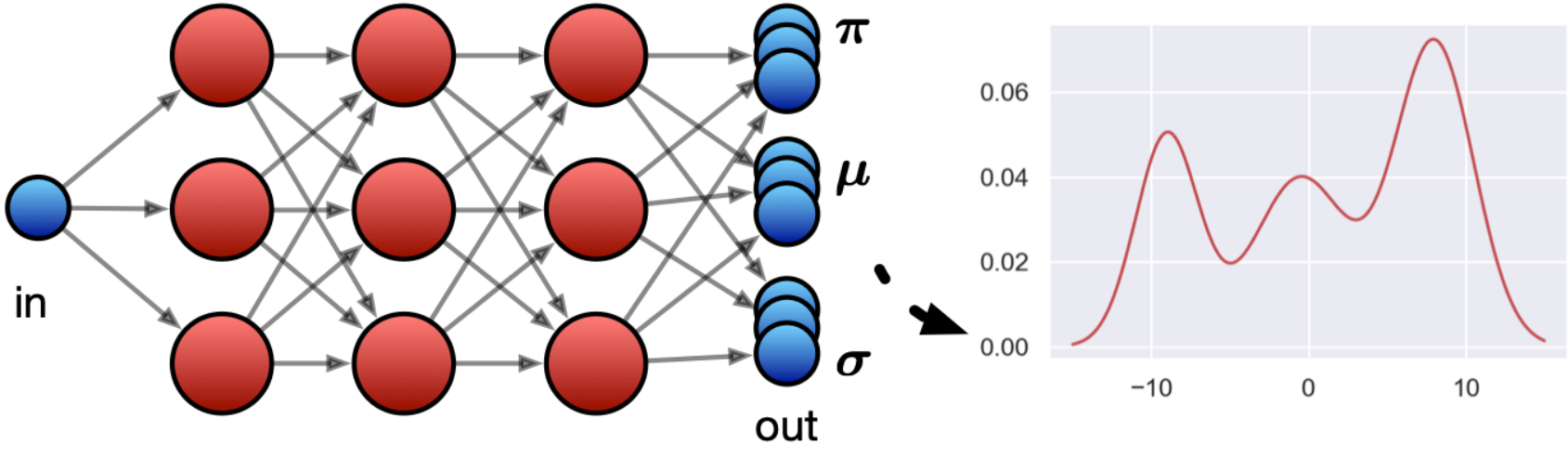




Here's a simple two-hidden-layer network (286 parameters), trained to produce the above result.

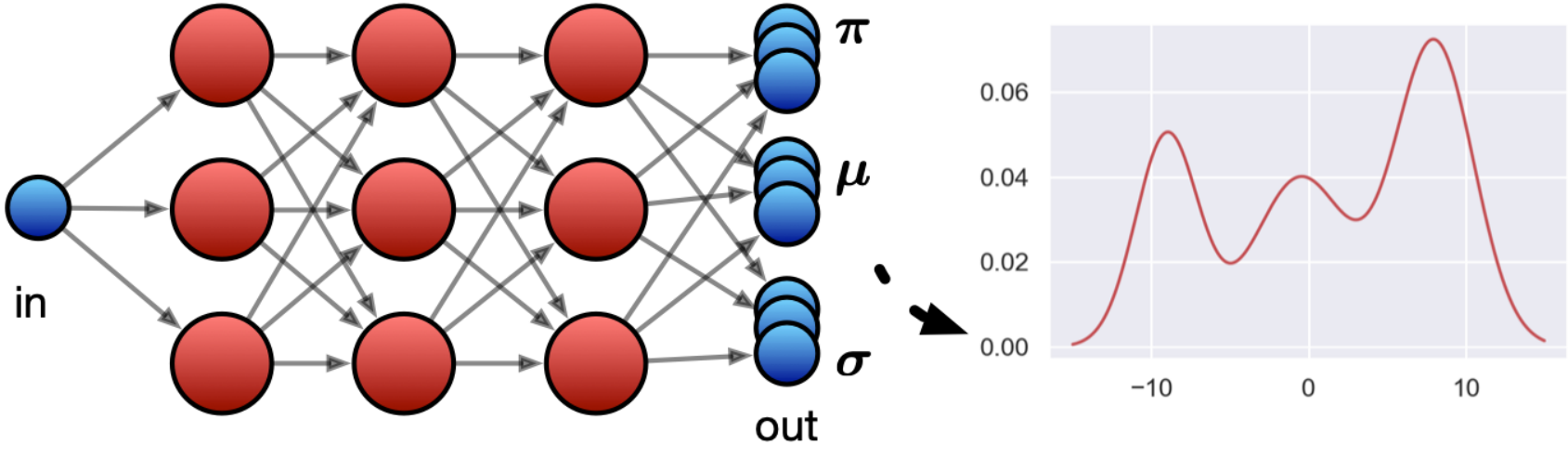
```
model = Sequential()
model.add(Dense(15, batch_input_shape=(None, 1), activation='tanh'))
model.add(Dense(15, activation='tanh'))
model.add(Dense(1, activation='linear'))
model.compile(loss='mse', optimizer='rmsprop')
model.fit(x=x_data, y=y_data, batch_size=128, epochs=200, validation_split=0.15)
```

MDN ARCHITECTURE:



$$\mathcal{L} = \sum_{i=1}^{K} \pi_i(\mathbf{x}) \mathcal{N}\left(\mu_i(\mathbf{x}), \sigma_i^2(\mathbf{x}); \mathbf{t}\right)$$

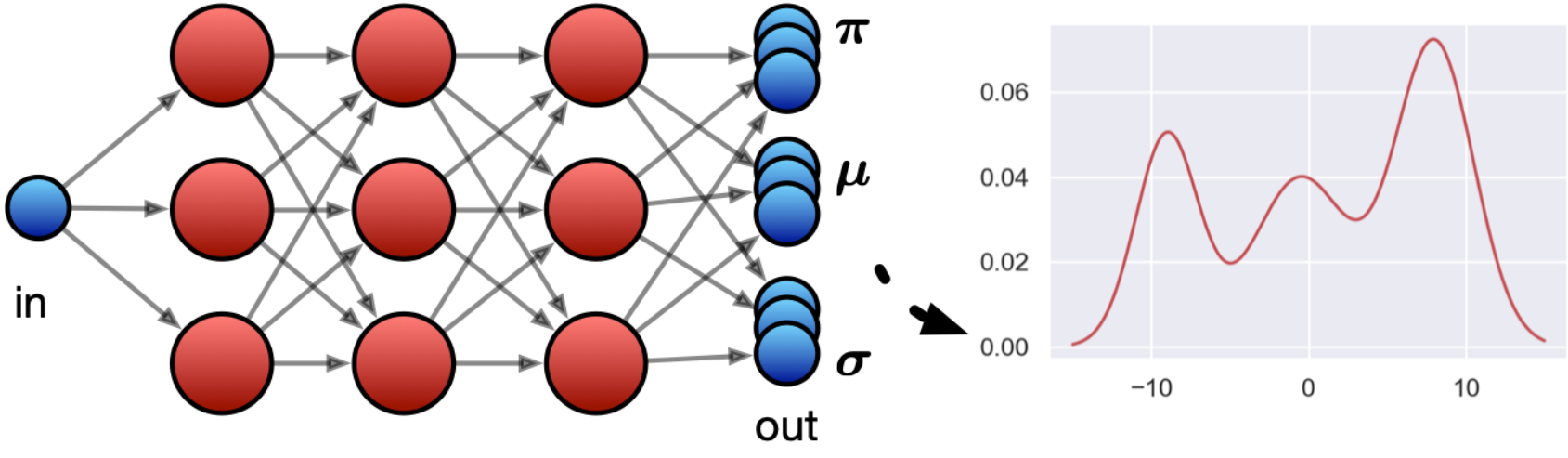
MDN ARCHITECTURE:



- Loss function for MDN is negative log of likelihood function L.

$$L = \sum_{i=1}^{K} \pi_i(x) N\left(\mu_i(x), \sigma_i^2(x); t\right)$$

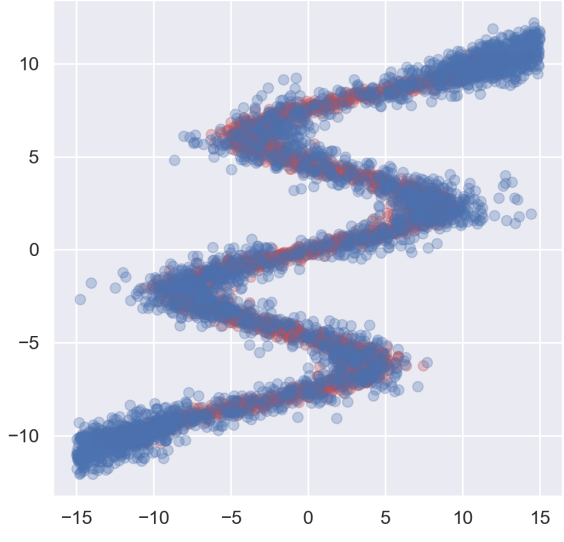
MDN ARCHITECTURE:

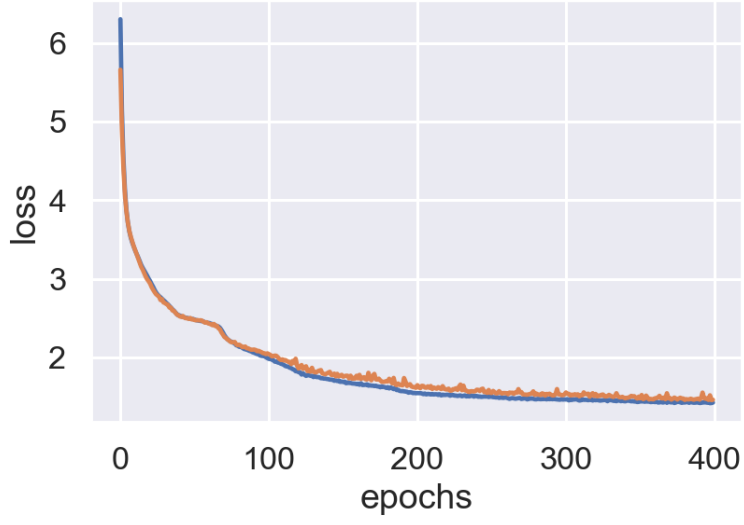


- Loss function for MDN is negative log of likelihood function L.
- L measures likelihood of t being drawn from a mixture parametrised by μ, σ, and π which are generated by the network inputs x:

$$L = \sum_{i=1}^{K} \pi_i(x) N\left(\mu_i(x), \sigma_i^2(x); t\right)$$

FEEDFORWARD MDN SOLUTION





And, here's a simple two-hidden-layer MDN (510 parameters), that achieves the above result! Much better!

```
N_MIXES = 5

model = Sequential()
model.add(Dense(15, batch_input_shape=(None, 1), activation='relu'))
model.add(Dense(15, activation='relu'))
model.add(mdn.MDN(1, N_MIXES)) # here's the MDN layer!
model.compile(loss=mdn.get_mixture_loss_func(1,N_MIXES), optimizer='rmsprop')
model.summary()
```

GETTING INSIDE THE MDN LAYER

Here's the same network wihtout using the MDN layer abstraction (this is with Keras' functional API):

```
def elu_plus_one_plus_epsilon(x):
    """ELU activation with a very small addition to help prevent NaN in loss."""
    return (K.elu(x) + 1 + 1e-8)

N_HIDDEN = 15
N_MIXES = 5

inputs = Input(shape=(1,), name='inputs')
hidden1 = Dense(N_HIDDEN, activation='relu', name='hidden1')(inputs)
hidden2 = Dense(N_HIDDEN, activation='relu', name='hidden2')(hidden1)

mdn_mus = Dense(N_MIXES, name='mdn_mus')(hidden2)
mdn_sigmas = Dense(N_MIXES, activation=elu_plus_one_plus_epsilon, name='mdn_sigmas')
  (hidden2)
mdn_pi = Dense(N_MIXES, name='mdn_pi')(hidden2)

mdn_out = Concatenate(name='mdn_outputs')([mdn_mus, mdn_sigmas, mdn_pi])

model = Model(inputs=inputs, outputs=mdn_out)
model.summary()
```

LOSS FUNCTION: THE TRICKY BIT.

Loss function for the MDN should be the negative log likelihood:

```
def mdn_loss(y_true, y_pred):
    # Split the inputs into paramaters
    out_mu, out_sigma, out_pi = tf.split(y_pred, num_or_size_splits=[N_MIXES, N_MIXES,
  N_MIXES],
                                         axis=-1, name='mdn_coef_split')
    mus = tf.split(out_mu, num_or_size_splits=N_MIXES, axis=1)
    sigs = tf.split(out_sigma, num_or_size_splits=N_MIXES, axis=1)
    # Construct the mixture models
    cat = tfd.Categorical(logits=out_pi)
    coll = [tfd.MultivariateNormalDiag(loc=loc, scale_diag=scale) for loc, scale
            in zip(mus, sigs)]
    mixture = tfd.Mixture(cat=cat, components=coll)
    # Calculate the loss function
    loss = mixture.log_prob(y_true)
    loss = tf.negative(loss)
    loss = tf.reduce_mean(loss)
    return loss

model.compile(loss=mdn_loss, optimizer='rmsprop')
```

Let’s go through bit by bit…

LOSS FUNCTION: PART 1:

First we have to extract the mixture paramaters.

```
# Split the inputs into paramaters
out_mu, out_sigma, out_pi = tf.split(y_pred, num_or_size_splits=[N_MIXES, N_MIXES, N_MIXES],
                                     axis=-1, name='mdn_coef_split')
mus = tf.split(out_mu, num_or_size_splits=N_MIXES, axis=1)
sigs = tf.split(out_sigma, num_or_size_splits=N_MIXES, axis=1)
```

LOSS FUNCTION: PART 1:

First we have to extract the mixture paramaters.

```
# Split the inputs into paramaters
out_mu, out_sigma, out_pi = tf.split(y_pred, num_or_size_splits=[N_MIXES, N_MIXES, N_MIXES],
                                     axis=-1, name='mdn_coef_split')
mus = tf.split(out_mu, num_or_size_splits=N_MIXES, axis=1)
sigs = tf.split(out_sigma, num_or_size_splits=N_MIXES, axis=1)
```

- Split up the parameters μ, σ, and π, remember that there are N_MIXES = K of each of these.

LOSS FUNCTION: PART 1:

First we have to extract the mixture paramaters.

```
# Split the inputs into paramaters
out_mu, out_sigma, out_pi = tf.split(y_pred, num_or_size_splits=[N_MIXES, N_MIXES, N_MIXES],
                                     axis=-1, name='mdn_coef_split')
mus = tf.split(out_mu, num_or_size_splits=N_MIXES, axis=1)
sigs = tf.split(out_sigma, num_or_size_splits=N_MIXES, axis=1)
```

- Split up the parameters μ, σ, and π, remember that there are N_MIXES = K of each of these.
- μ and σ have to be split *again* so that we can iterate over them (you can't iterate over an axis of a tensor…)

LOSS FUNCTION: PART 2:

Now we have to construct the mixture model's PDF.

```
# Construct the mixture models
cat = tfd.Categorical(logits=out_pi)
coll = [tfd.Normal(loc=loc, scale=scale) for loc, scale
        in zip(mus, sigs)]
mixture = tfd.Mixture(cat=cat, components=coll)
```

LOSS FUNCTION: PART 2:

Now we have to construct the mixture model's PDF.

```
# Construct the mixture models
cat = tfd.Categorical(logits=out_pi)
coll = [tfd.Normal(loc=loc, scale=scale) for loc, scale
        in zip(mus, sigs)]
mixture = tfd.Mixture(cat=cat, components=coll)
```

- For this, we're using the `Mixture` abstraction provided in `tensorflow-probability.distributions`.

LOSS FUNCTION: PART 2:

Now we have to construct the mixture model's PDF.

```
# Construct the mixture models
cat = tfd.Categorical(logits=out_pi)
coll = [tfd.Normal(loc=loc, scale=scale) for loc, scale
        in zip(mus, sigs)]
mixture = tfd.Mixture(cat=cat, components=coll)
```

- For this, we're using the `Mixture` abstraction provided in `tensorflow-probability.distributions`.
- This takes a categorical (a.k.a. softmax, a.k.a. generalized Bernoulli distribution) model, and a list the component distributions.

LOSS FUNCTION: PART 2:

Now we have to construct the mixture model's PDF.

```
# Construct the mixture models
cat = tfd.Categorical(logits=out_pi)
coll = [tfd.Normal(loc=loc, scale=scale) for loc, scale
        in zip(mus, sigs)]
mixture = tfd.Mixture(cat=cat, components=coll)
```

- For this, we're using the `Mixture` abstraction provided in `tensorflow-probability.distributions`.
- This takes a categorical (a.k.a. softmax, a.k.a. generalized Bernoulli distribution) model, and a list the component distributions.
- Each normal PDF is contructed using `tfd.Normal`.

LOSS FUNCTION: PART 2:

Now we have to construct the mixture model's PDF.

```
# Construct the mixture models
cat = tfd.Categorical(logits=out_pi)
coll = [tfd.Normal(loc=loc, scale=scale) for loc, scale
        in zip(mus, sigs)]
mixture = tfd.Mixture(cat=cat, components=coll)
```

- For this, we're using the `Mixture` abstraction provided in `tensorflow-probability.distributions`.
- This takes a categorical (a.k.a. softmax, a.k.a. generalized Bernoulli distribution) model, and a list the component distributions.
- Each normal PDF is contructed using `tfd.Normal`.
- Can do this from first principles as well, but good to use abstractions that are available (?)

LOSS FUNCTION: PART 3:

Finally, we calculate the loss:

```
loss = mixture.log_prob(y_true)
loss = tf.negative(loss)
loss = tf.reduce_mean(loss)
```

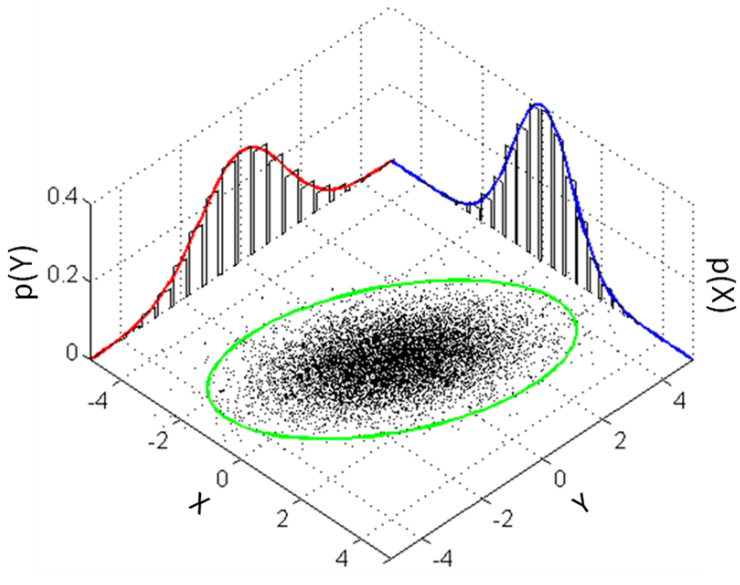
LOSS FUNCTION: PART 3:

Finally, we calculate the loss:

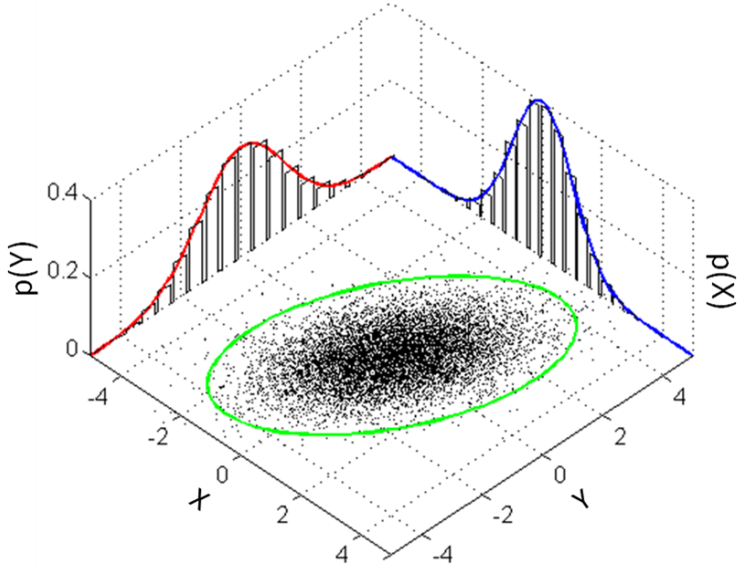
```
loss = mixture.log_prob(y_true)
loss = tf.negative(loss)
loss = tf.reduce_mean(loss)
```

- `mixture.log_prob(y_true)` means “the log-likelihood of sampling `y_true` from the distribution called `mixture`.”

SOME MORE DETAILS….

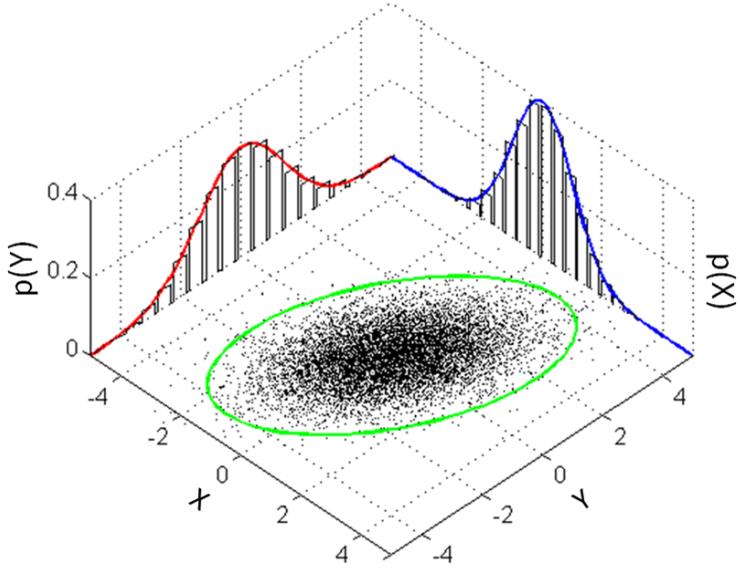


SOME MORE DETAILS….



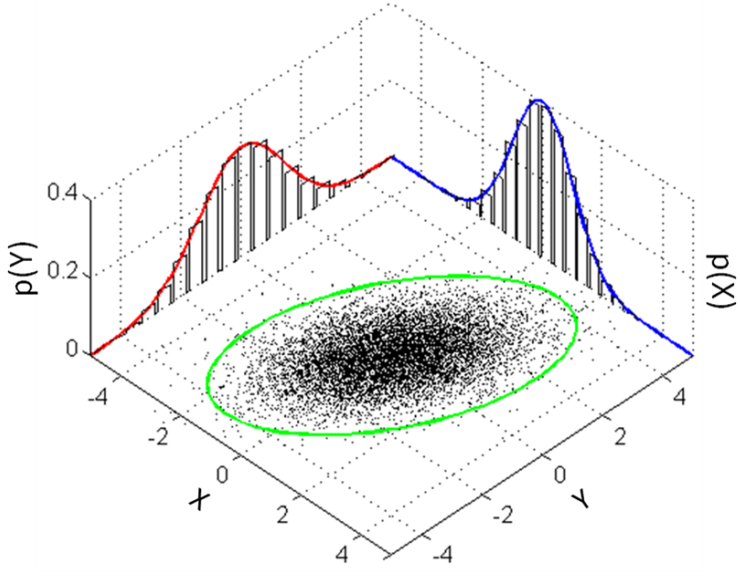
- This “version” of a mixture model works for a mixture of 1D normal distributions.

SOME MORE DETAILS….



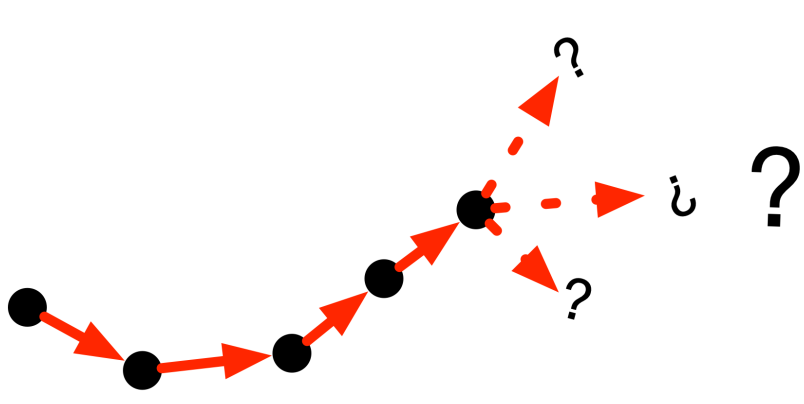
- This “version” of a mixture model works for a mixture of 1D normal distributions.
- Not too hard to extend to multivariate normal distributions, which are useful for lots of problems.

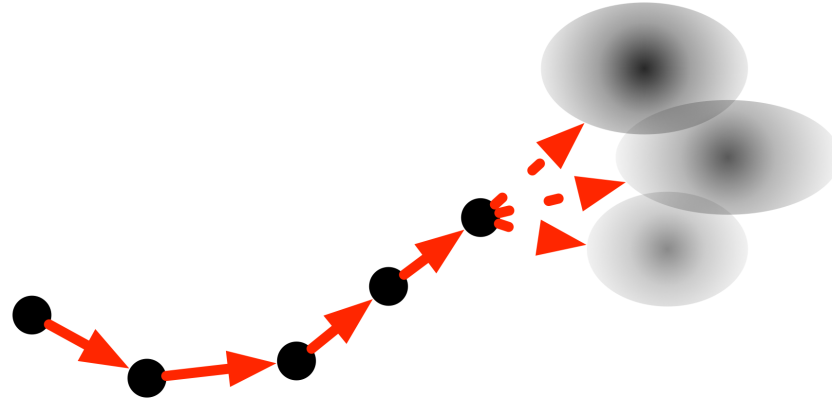
SOME MORE DETAILS….



- This “version” of a mixture model works for a mixture of 1D normal distributions.
- Not too hard to extend to multivariate normal distributions, which are useful for lots of problems.
- This is how it actually works in my Keras MDN layer, have a look at the code for more details…

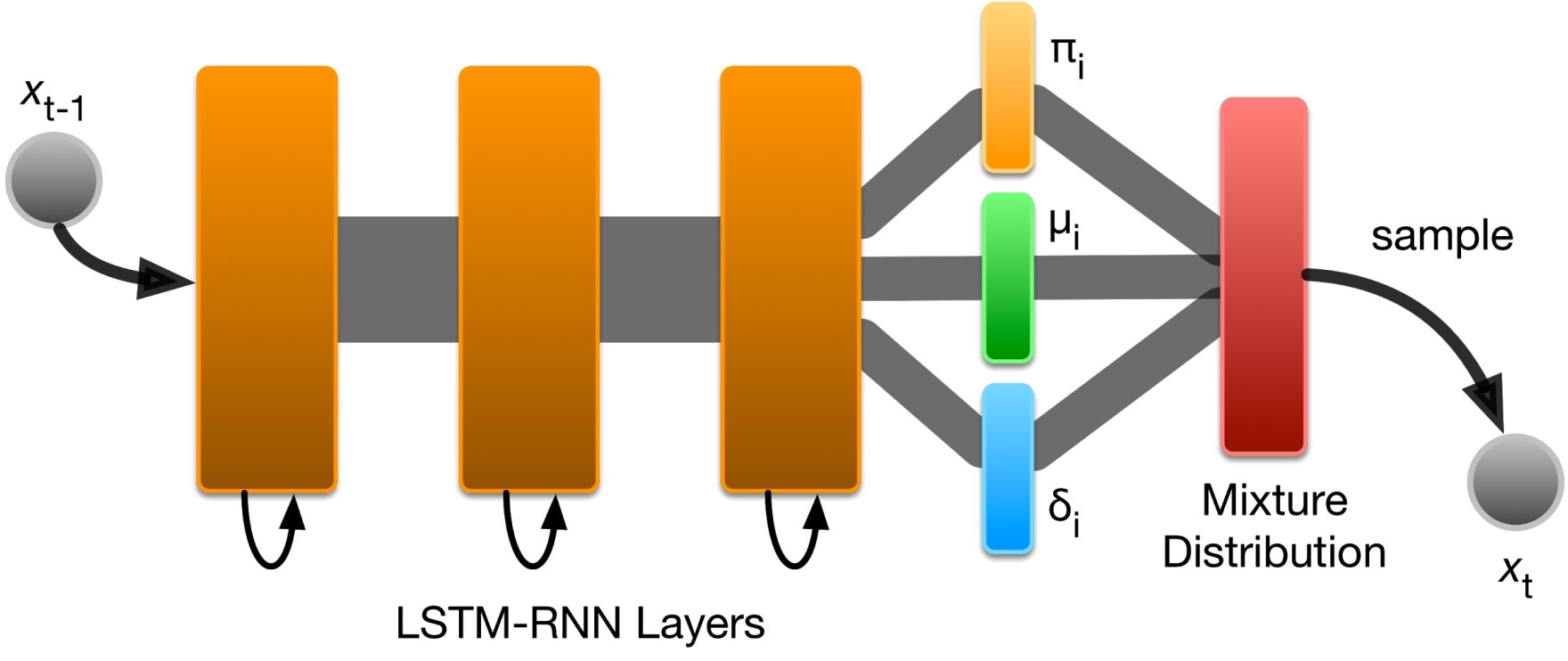
MDN-RNNS





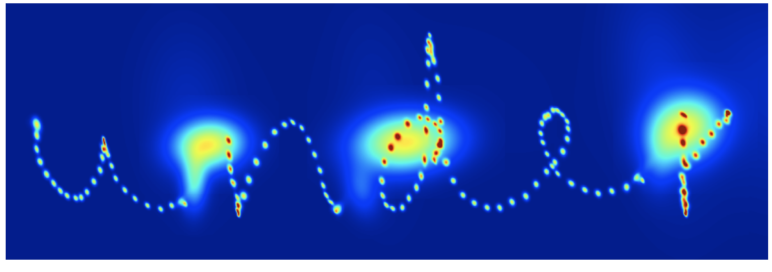
MDNs can be handy at the end of an RNN! Imagine a robot calculating moves forward through space, it might have to choose from a number of valid positions, each of which could be modelled by a 2D Normal model.

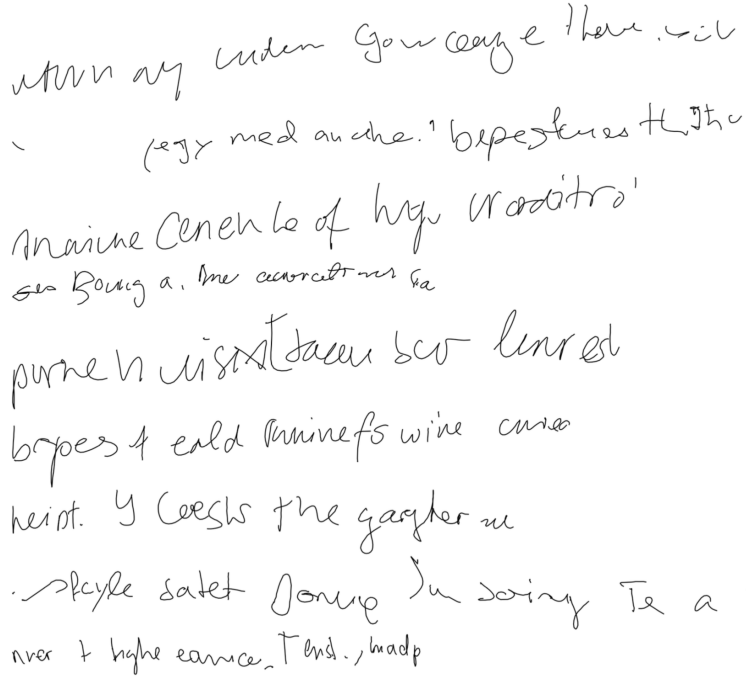
MDN-RNN ARCHITECTURE



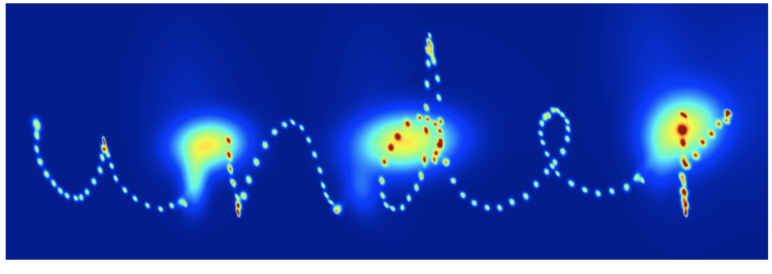
Can be as simple as putting an MDN layer after recurrent layers!

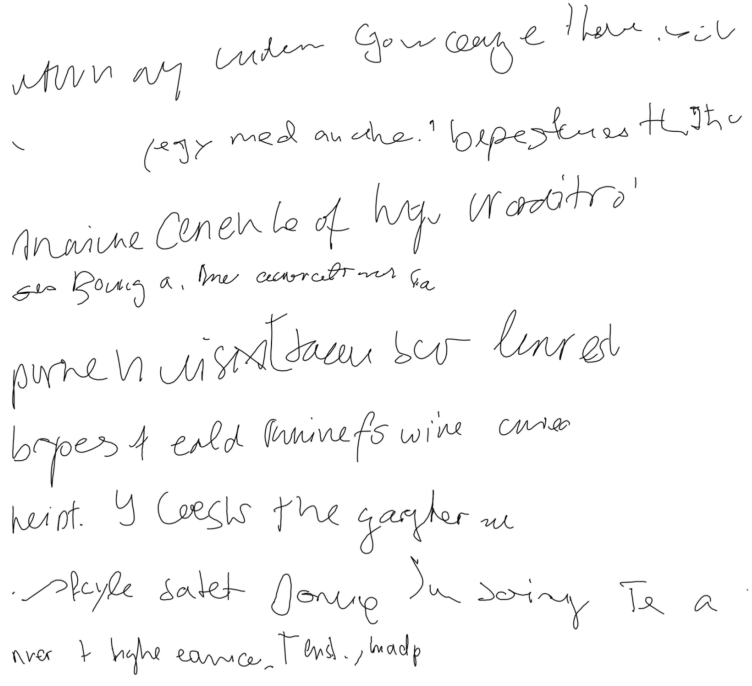
USE CASES: HANDWRITING GENERATION





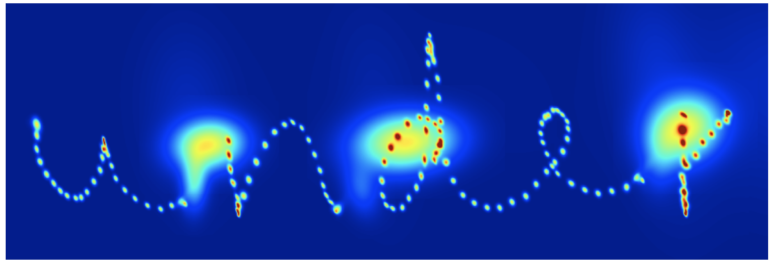
USE CASES: HANDWRITING GENERATION

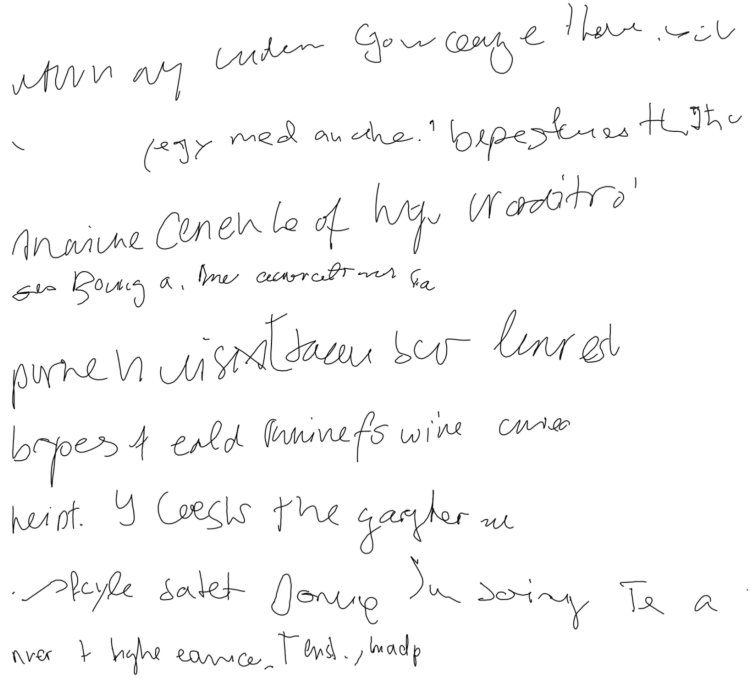




- Handwriting Generation RNN (Graves, 2013).

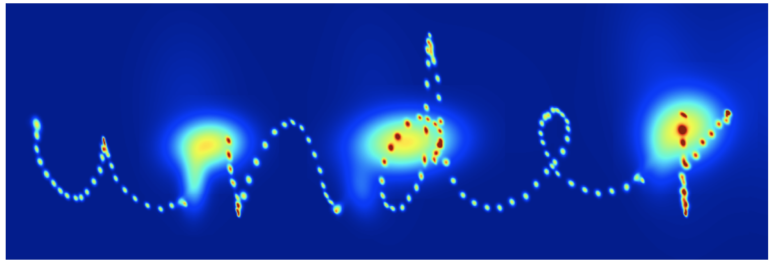
USE CASES: HANDWRITING GENERATION

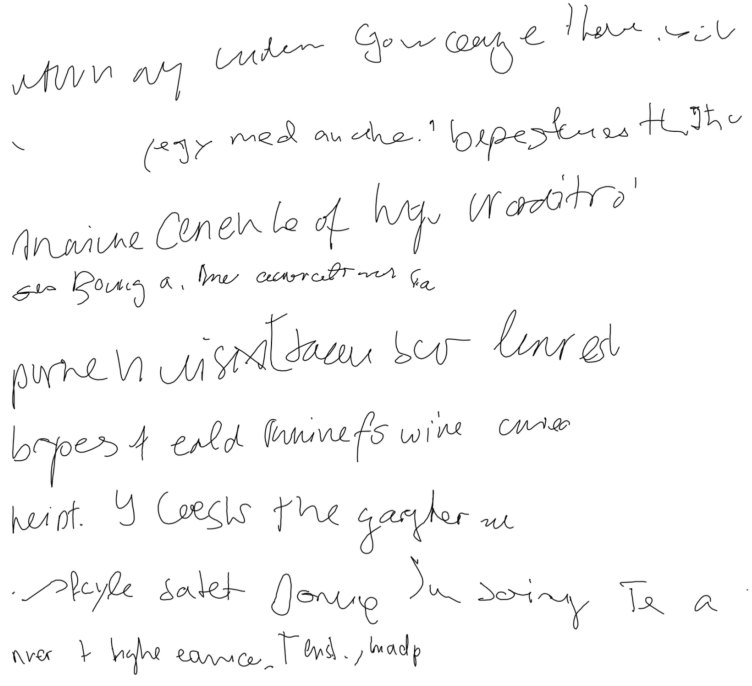




- Handwriting Generation RNN (Graves, 2013).
- Trained on handwriting data.

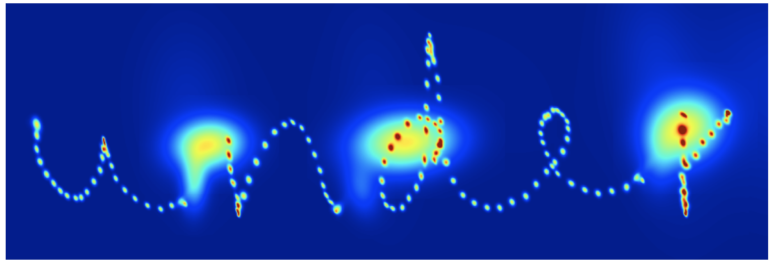
USE CASES: HANDWRITING GENERATION

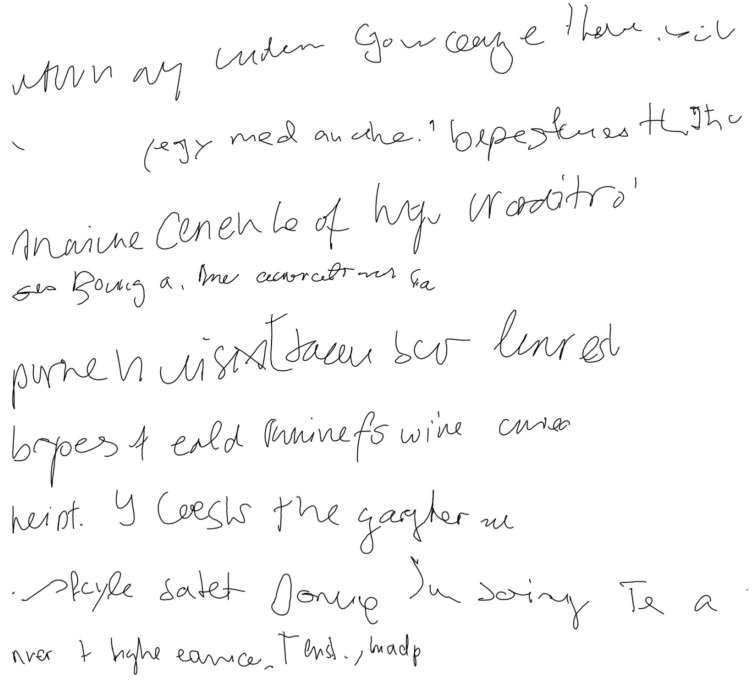




- Handwriting Generation RNN (Graves, 2013).
- Trained on handwriting data.
- Predicts the next location of the pen (dx, dy, and up/down)

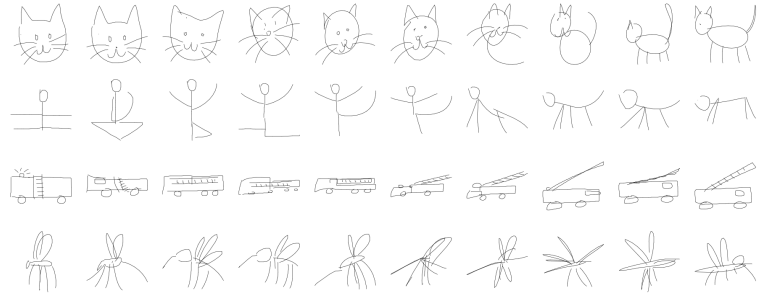
USE CASES: HANDWRITING GENERATION





- Handwriting Generation RNN (Graves, 2013).
- Trained on handwriting data.
- Predicts the next location of the pen (dx, dy, and up/down)
- Network takes text to write as an extra input, RNN learns to decide what character to write next.

USE CASES: SKETCHRNN



USE CASES: SKETCHRNN

- SketchRNN Kanji (Ha, 2015); similar to handwriting generation, trained on kanji and then generates new “fake” characters

USE CASES: SKETCHRNN

- SketchRNN Kanji (Ha, 2015); similar to handwriting generation, trained on kanji and then generates new “fake” characters
- SketchRNN VAE (Ha et al., 2017); similar again, but trained on human-sourced sketches. VAE architecture with bidirectional RNN encoder and MDN in the decoder part.

USE CASES: ROBOJAM

USE CASES: ROBOJAM

- RoboJam (Martin et al., 2018); similar to the kanji RNN, but trained on touchscreen musical performances

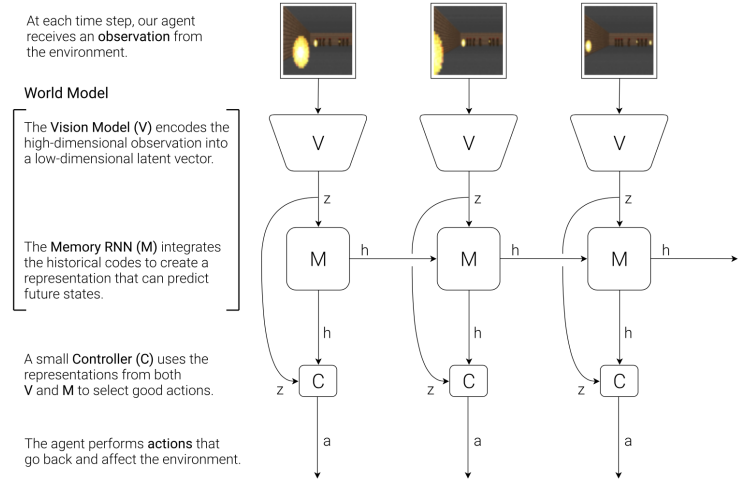
USE CASES: ROBOJAM

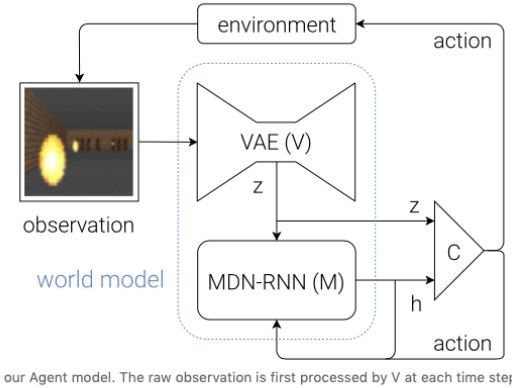
- RoboJam (Martin et al., 2018); similar to the kanji RNN, but trained on touchscreen musical performances
- Extra complexity: have to model touch position (x, y) and time (dt).

USE CASES: ROBOJAM

- RoboJam (Martin et al., 2018); similar to the kanji RNN, but trained on touchscreen musical performances
- Extra complexity: have to model touch position (x, y) and time (dt).
- Implemented in my MicroJam app (have a go: microjam.info)

USE CASES: WORLD MODELS

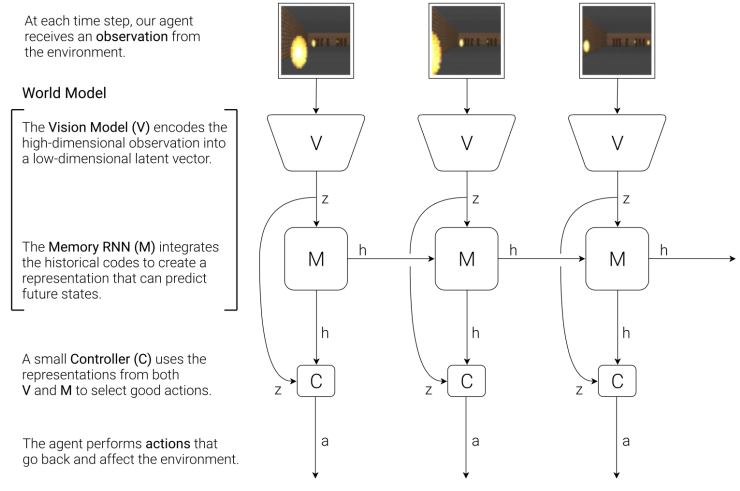


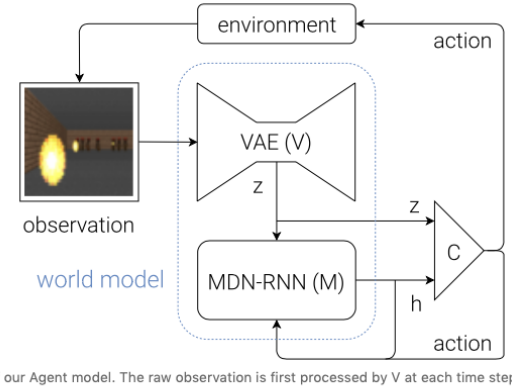


Flow diagram of our Agent model. The raw observation is first processed by V at each time step t to produce z_t. The input into C is this latent vector z_t concatenated with M's hidden state h_t at each time step. C will then output an action vector a_t for motor control. M will then take the current z_t and action a_t as an input to update its own hidden state to produce h_{t+1} to be used at time $t+1$.

USE CASES: WORLD MODELS

- World Models (Ha & Schmidhuber, 2018)

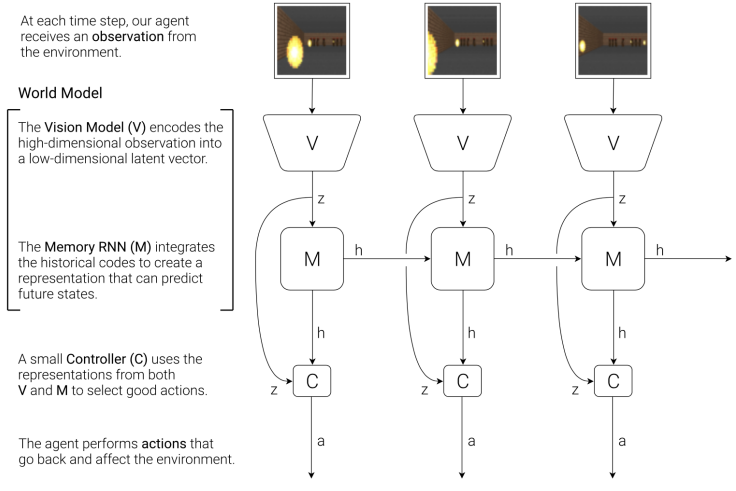


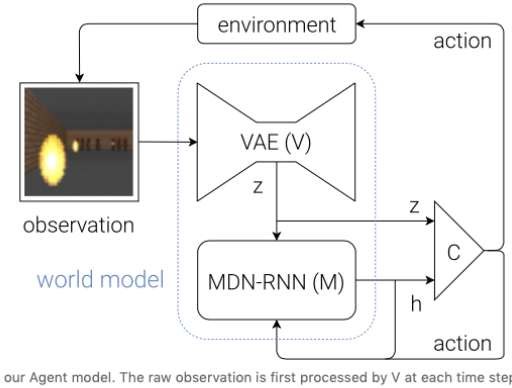


Flow diagram of our Agent model. The raw observation is first processed by V at each time step t to produce z_t. The input into C is this latent vector z_t concatenated with M's hidden state h_t at each time step. C will then output an action vector a_t for motor control. M will then take the current z_t and action a_t as an input to update its own hidden state to produce h_{t+1} to be used at time $t+1$.

USE CASES: WORLD MODELS

- World Models (Ha & Schmidhuber, 2018)
- Train a VAE for visual perception an environment (e.g., VizDoom), now each frame from the environment can be represented by a vector z

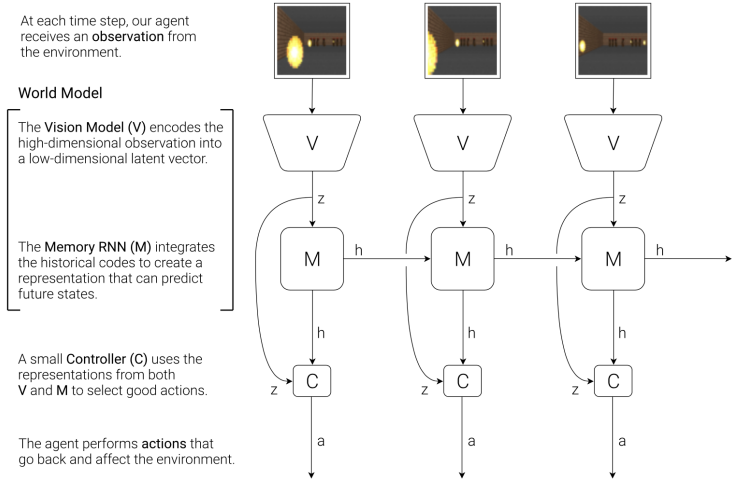


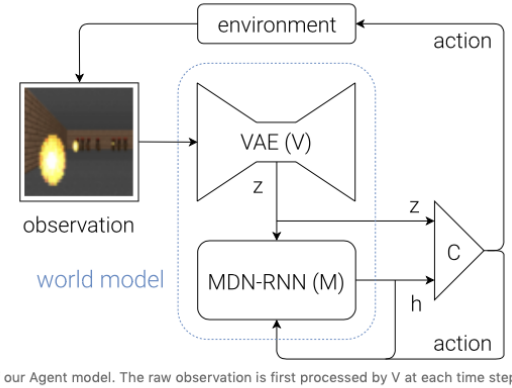


Flow diagram of our Agent model. The raw observation is first processed by V at each time step t to produce z_t. The input into C is this latent vector z_t concatenated with M's hidden state h_t at each time step. C will then output an action vector a_t for motor control. M will then take the current z_t and action a_t as an input to update its own hidden state to produce h_{t+1} to be used at time $t+1$.

USE CASES: WORLD MODELS

- World Models (Ha & Schmidhuber, 2018)
- Train a VAE for visual perception an environment (e.g., VizDoom), now each frame from the environment can be represented by a vector z
- Train MDN to predict next z, use this to help train an agent to operate in the environment.





Flow diagram of our Agent model. The raw observation is first processed by V at each time step t to produce z_t. The input into C is this latent vector z_t concatenated with M's hidden state h_t at each time step. C will then output an action vector a_t for motor control. M will then take the current z_t and action a_t as an input to update its own hidden state to produce h_{t+1} to be used at time $t+1$.

REFERENCES

REFERENCES

1. Christopher M. Bishop. 1994. Mixture Density Networks. Technical Report NCRG/94/004. Neural Computing Research Group, Aston University.

REFERENCES

1. Christopher M. Bishop. 1994. Mixture Density Networks. Technical Report NCRG/94/004. Neural Computing Research Group, Aston University.
2. Axel Brando. 2017. Mixture Density Networks (MDN) for distribution and uncertainty estimation. Master's thesis. Universitat Politècnica de Catalunya.

REFERENCES

1. Christopher M. Bishop. 1994. Mixture Density Networks. Technical Report NCRG/94/004. Neural Computing Research Group, Aston University.
2. Axel Brando. 2017. Mixture Density Networks (MDN) for distribution and uncertainty estimation. Master's thesis. Universitat Politècnica de Catalunya.
3. A. Graves. 2013. Generating Sequences With Recurrent Neural Networks. ArXiv e-prints (Aug. 2013). ArXiv:1308.0850

REFERENCES

1. Christopher M. Bishop. 1994. Mixture Density Networks. Technical Report NCRG/94/004. Neural Computing Research Group, Aston University.
2. Axel Brando. 2017. Mixture Density Networks (MDN) for distribution and uncertainty estimation. Master's thesis. Universitat Politècnica de Catalunya.
3. A. Graves. 2013. Generating Sequences With Recurrent Neural Networks. ArXiv e-prints (Aug. 2013). ArXiv:1308.0850
4. David Ha and Douglas Eck. 2017. A Neural Representation of Sketch Drawings. ArXiv e-prints (April 2017). ArXiv:1704.03477

REFERENCES

1. Christopher M. Bishop. 1994. Mixture Density Networks. Technical Report NCRG/94/004. Neural Computing Research Group, Aston University.
2. Axel Brando. 2017. Mixture Density Networks (MDN) for distribution and uncertainty estimation. Master's thesis. Universitat Politècnica de Catalunya.
3. A. Graves. 2013. Generating Sequences With Recurrent Neural Networks. ArXiv e-prints (Aug. 2013). ArXiv:1308.0850
4. David Ha and Douglas Eck. 2017. A Neural Representation of Sketch Drawings. ArXiv e-prints (April 2017). ArXiv:1704.03477
5. Charles P. Martin and Jim Torresen. 2018. RoboJam: A Musical Mixture Density Network for Collaborative Touchscreen Interaction. In Evolutionary and Biologically Inspired Music, Sound, Art and Design: EvoMUSART '18, A. Liapis et al. (Ed.). Lecture Notes in Computer Science, Vol. 10783. Springer International Publishing. DOI:10.1007/9778-3-319-77583-8_11

REFERENCES

1. Christopher M. Bishop. 1994. Mixture Density Networks. Technical Report NCRG/94/004. Neural Computing Research Group, Aston University.
2. Axel Brando. 2017. Mixture Density Networks (MDN) for distribution and uncertainty estimation. Master's thesis. Universitat Politècnica de Catalunya.
3. A. Graves. 2013. Generating Sequences With Recurrent Neural Networks. ArXiv e-prints (Aug. 2013). ArXiv:1308.0850
4. David Ha and Douglas Eck. 2017. A Neural Representation of Sketch Drawings. ArXiv e-prints (April 2017). ArXiv:1704.03477
5. Charles P. Martin and Jim Torresen. 2018. RoboJam: A Musical Mixture Density Network for Collaborative Touchscreen Interaction. In Evolutionary and Biologically Inspired Music, Sound, Art and Design: EvoMUSART '18, A. Liapis et al. (Ed.). Lecture Notes in Computer Science, Vol. 10783. Springer International Publishing. DOI:10.1007/9778-3-319-77583-8_11
6. D. Ha and J. Schmidhuber. 2018. Recurrent World Models Facilitate Policy Evolution. ArXiv e-prints (Sept. 2018). ArXiv:1809.01999