# Week 11: Neural Networks for Text

POP77032 Quantitative Text Analysis for Social Scientists

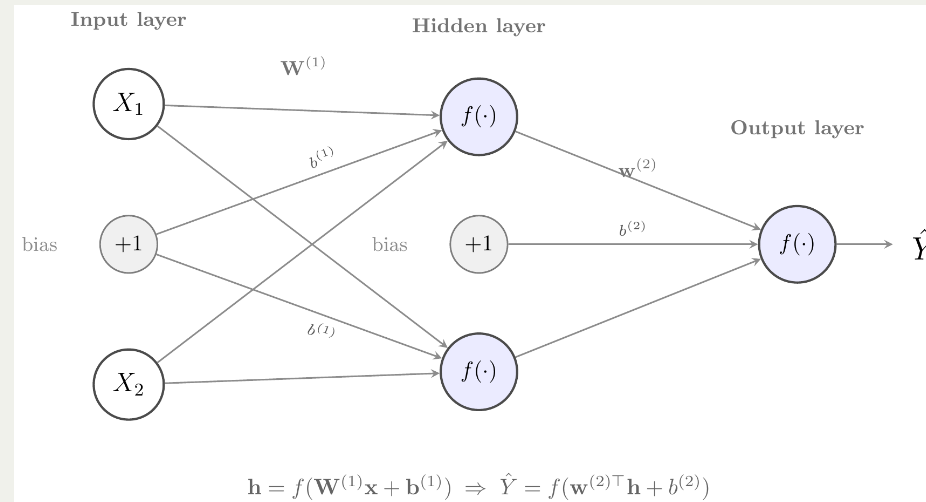
Tom Paskhalis

# Overview

- Feedforward neural network (MLP)
- Backpropagation
- Deep learning libraries
- Neural networks for text

# Deep Feedforward Neural Networks

# Recap: MLP (Deep Feedforward Neural Network)




- A **multi-layer perceptron (MLP)** adds one or more **hidden layers** between the inputs and the output.
- Each hidden neuron applies an activation function $f$ to its weighted inputs, learning non-linear intermediate representations.
- With a hidden layer and non-linear activation, an MLP can represent **non-linearly separable** functions such as XOR.

# MLP for XOR

```
# Rectified Linear Unit (ReLU)
def relu(z):
    return np.maximum(0, z)

# Sigmoid activation functions
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([0, 1, 1, 0])

rng = np.random.default_rng(1)
W1 = rng.normal(scale=0.5, size=(2, 2)) # hidden weights (2×2)
b1 = np.zeros(2) # hidden biases
w2 = rng.normal(scale=0.5, size=2) # output weights
b2 = 0.0 # output bias
lr = 0.1
batch_size = 10
num_epochs = 1000

for epoch in range(1000):
    indices = np.random.choice(len(y), size=batch_size, replace=True)
    X_batch, y_batch = X[indices], y[indices]
    z = X_batch @ W1 + b1          # hidden pre-activation
    h  = relu(z)                    # hidden layer (ReLU)
    y_pred = sigmoid(h @ w2 + b2)   # output (sigmoid)
    residuals = y_batch - y_pred
    sig_d = y_pred * (1 - y_pred)   # sigmoid derivative
    delta = np.outer(residuals * sig_d, w2) * (z > 0) # backpropagation
    gradient_W1 = -2 * X_batch.T @ delta / batch_size
    gradient_b1 = -2 * np.sum(delta, axis=0) / batch_size
    gradient_w2 = -2 * h.T @ (residuals * sig_d) / batch_size
    gradient_b2 = -2 * np.sum(residuals * sig_d) / batch_size
    W1 -= lr * gradient_W1;  b1 -= lr * gradient_b1
    w2 -= lr * gradient_w2;  b2 -= lr * gradient_b2
```

```
array([0.194768  , 0.8675337 , 0.89020838, 0.19471946])
```

# Forward Propagation

- In **deep feedforward neural network** (aka multi-layer perceptron) information flows in the direction from the input layer $\mathbf{x}$ to the output layer $\hat{\mathbf{y}}$.
- This information flow is called **forward propagation**.
- Remember that the flow of information is, essentially, done through calculating the derivatives of the output with respect to the inputs and parameters (gradients).
- For a SLP which could be represented as simply $y = f(x)$, where $f()$ is some activation function, this is quite straightforward:

$$f'(x) = \frac{dy}{dx}$$

# Function Composition

- Things, however, get a bit more complicated when we start adding hidden layers to our network.
- Each intermediate (hidden) layer is, effectively, another function applied to the output of the previous function.
- Recall our discussion of function composition and nested functions from POP77001.
- The output of the overall network (actual prediction) then becomes:

$$y = f(g(x)) = f(z)$$

# Chain Rule of Calculus

- As often, calculus comes to our rescue.
- If we represent the output as $y = f(g(x)) = f(z)$, than the chain rules states that:

$$f'(g(x)) = f'(g(x)) \cdot g'(x)$$

- Or, alternatively:

$$\frac{dy}{dx} = \frac{dy}{dz} \cdot \frac{dz}{dx}$$

- Which can be generalised to any number of nested functions (i.e., hidden layers):

$$\frac{dy}{dx} = \frac{dy}{dz_1} \cdot \frac{dz_1}{dz_2} \cdot \ldots \cdot \frac{dz_{n-1}}{dx}$$

# Backpropagation

- Further generalising beyond the scalar case, provides us with the mathematical basis for the **backpropagation** algorithm.
- The idea behind backpropagation is to look at ancestors of some node in the computational graph and compute their gradients by applying the chain rule.
- If there is more than one path from the ancestor to the node, we need to sum over all paths.
- In principle, backpropagation is a general algorithm that can be applied to any computational graph, not just neural network.
- In practice, it is most commonly used for training neural networks, where the computational graph can be quite complex due to multiple layers and non-linear activations.

# Example: Backpropagation for XOR

- Let's see how backpropagation works in practice by looking at the XOR example we implemented earlier.
- Assuming we use NLL (a more appropriate loss function for binary classification), the output layer computes the following gradient:

$$\frac{\partial \text{NLL}}{\partial w^{(2)}} = -\sum_{i=1}^{N} (y_i - \hat{y}_i) h_i$$

$$\frac{\partial \text{NLL}}{\partial b^{(2)}} = -\sum_{i=1}^{N} (y_i - \hat{y}_i)$$

where $h_i$ is the output of the hidden layer, and $\hat{y}_i = \sigma(z) = \sigma(h_i^T w^{(2)} + b^{(2)})$

- From the tutorial recall that the derivative of the sigmoid function is given by:

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

# Example: Backpropagation for XOR

- Since we applied the ReLU activation function in the hidden layer, we need to compute its derivative as well:

$$\text{ReLU}'(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases}$$

- Putting both of these together, we can compute the gradient for the hidden layer using the chain rule:

$$\frac{\partial \text{NLL}}{\partial W^{(1)}} = -\sum_{i=1}^{N} (y_i - \hat{y}_i) w^{(2)} \cdot \mathbf{1}_{z_i > 0} \cdot X_i$$

$$\frac{\partial \text{NLL}}{\partial b^{(1)}} = -\sum_{i=1}^{N} (y_i - \hat{y}_i) w^{(2)} \cdot \mathbf{1}_{z_i > 0}$$

- When implemented in code the last equations would look like this:

```
delta = np.outer(residuals, w2) * (z > 0) # backpropagation
gradient_W1 = -X_batch.T @ delta / batch_size
gradient_b1 = -np.sum(delta, axis=0) / batch_size
```

# MLP for XOR with NLL Loss

```
1  # Rectified Linear Unit (ReLU)
2  def relu(z):
3      return np.maximum(0, z)
4
5  # Sigmoid activation functions
6  def sigmoid(z):
7      return 1 / (1 + np.exp(-z))
8
9  X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
10 y = np.array([0, 1, 1, 0])
11
12 rng = np.random.default_rng(1)
13 W1 = rng.normal(scale=0.5, size=(2, 2)) # hidden weights (2×2)
14 b1 = np.zeros(2) # hidden biases
15 w2 = rng.normal(scale=0.5, size=2) # output weights
16 b2 = 0.0 # output bias
17 lr = 0.1
18 batch_size = 10
19 num_epochs = 1000
20
21 for epoch in range(1000):
22     indices = np.random.choice(len(y), size=batch_size, replace=True)
23     X_batch, y_batch = X[indices], y[indices]
24     z = X_batch @ W1 + b1          # hidden pre-activation
25     h  = relu(z)                    # hidden layer (ReLU)
26     y_pred = sigmoid(h @ w2 + b2)   # output (sigmoid)
27     residuals = y_batch - y_pred
28     delta = np.outer(residuals, w2) * (z > 0) # backpropagation
29     gradient_W1 = -X_batch.T @ delta / batch_size
30     gradient_b1 = -np.sum(delta, axis=0) / batch_size
31     gradient_w2 = -h.T @ (residuals * (y_pred * (1 - y_pred))) / batch_size
32     gradient_b2 = -np.sum(residuals * (y_pred * (1 - y_pred))) / batch_size
33     W1 -= lr * gradient_W1;  b1 -= lr * gradient_b1
34     w2 -= lr * gradient_w2;  b2 -= lr * gradient_b2
35 sigmoid(relu(X @ W1 + b1) @ w2 + b2)
```

```
array([0.20078052, 0.91266064, 0.91991984, 0.20078052])
```

# Objective Function

- The function that we are trying to optimise is our **objective function**.
- In the case of SGD, we are always *minimising* that function (hence, **loss function**).
- With other estimation methods (e.g. MLE) you might be maximising some function (e.g., likelihood).
- But you can always invert the maximisation problem into a minimisation problem by negating the function.
- Thus, maximising the likelihood is equivalent to minimising the negative log-likelihood (NLL).

# Objective Functions

- So far, we have seen MSE and NLL used as objective functions.
- But there are other objective functions that are often used in neural networks.
- Some common examples include:
  - **Absolute Error (MAE)** (aka L1 loss): $\frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|$
  - **Squared Error (MSE)** (aka L2 loss): $\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$
  - **Kullback-Leibler (KL) divergence**: $\sum_{i=1}^{N} y_i \log\left(\frac{y_i}{\hat{y}_i}\right)$
  - **Cross-Entropy** and closely related to it **Negative Log-Likelihood (NLL)**: $-\sum_{i=1}^{N} y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)$

# Choosing Objective Function

- While the values of objective functions can be different, the optimal $\theta$ is always the same.
- Some aspects to consider when choosing an objective function include:
  - **Regression vs. Classification**: For problems with discrete labels, NLL/Cross-Entropy is more appropriate than MSE.
  - **Interpretability**: Some objective functions (e.g., MAE) are more interpretable than others (e.g., KL divergence).
  - **Computational efficiency**: Some objective functions are easier to compute and differentiate than others.
  - **Robustness to outliers**: Some objective functions (e.g., MAE) are more robust to outliers than others (e.g., MSE).

# Learning Rate

- The **learning rate** ($\epsilon$) is a crucial hyperparameter that is used by SGD.
- It has a significant impact on model performance while being difficult to tune.
- While ordinary gradient descent can work with a fixed learning rate, when applying SGD, it is necessary to decay (decrease) the learning rate over time to ensure convergence.
- As SGD estimator introduces noise through random sampling it does not become 0 even at a minimum.
- Common learning rate decay schedules include:
  - **Step decay**: Reduce the learning rate by a factor every few epochs (e.g., halve it every 10 epochs).
  - **Exponential decay**: Multiply the learning rate by a factor (e.g., 0.9) after each epoch.
  - **Adaptive methods**: Algorithms like Adam adjust the learning rate dynamically based on the gradients.

# Example: Learning Rate Decay

- Let's modify the SGD algorithm that we implemented for linear regression last week.

```
import numpy as np

rng = np.random.default_rng(seed = 123)
X = rng.normal(size = (1000, 2))
y = 1.5 * X[:, 0] - 2.0 * X[:, 1] + rng.normal(size = 1000)
```

```
params = np.zeros(2)  # initial weights (betas)
lr = 0.01 # learning rate
batch_size = 50
num_epochs = 1000

for epoch in range(num_epochs):
    indices = np.random.choice(len(y), size=batch_size, replace=False)
    X_batch, y_batch = X[indices], y[indices]

    y_pred = X_batch @ params
    residuals = y_batch - y_pred
    gradient_params = -2 * X_batch.T @ residuals / batch_size
    params = params - lr * gradient_params

params
```

```
array([ 1.48110149, -2.048901  ])
```

# Example: Exponential Decay

- A simple way to introduce learning rate decay is to multiply the learning rate by some factor after each epoch:

```
params = np.zeros(2)  # initial weights (betas)
lr = 0.01 # learning rate
batch_size = 50
num_epochs = 1000

for epoch in range(num_epochs):
    indices = np.random.choice(len(y), size=batch_size, replace=False)
    X_batch, y_batch = X[indices], y[indices]
    lr = lr * 0.9  # exponential decay

    y_pred = X_batch @ params
    residuals = y_batch - y_pred
    gradient_params = -2 * X_batch.T @ residuals / batch_size
    params = params - lr * gradient_params

params
```

```
array([ 0.26061463, -0.36474975])
```

# Adaptive Learning Rates

- Incorporating learning rate decay is, essentially, adding another hyperparameter that needs to be tuned.
- In practice, most use **adaptive learning rate optimisers** that adjust the learning rate dynamically based on the gradients.
- These tend to be based on certain heuristics about the behaviour of the gradients during training:
  - Gradient consistency across minibatches.
  - Higher loss sensitivity to some parameters than others.

# Adaptive Learning Rate Optimisation Algorithms

- Some popular examples of adaptive learning rate optimisation algorithms include:
  - **Adagrad**: Adapts the learning rate for each parameter based on the historical sum of squared gradients.
  - **RMSprop**: Similar to Adagrad but uses a moving average of squared gradients to prevent the learning rate from decaying too much.
  - **Adam**: Combines the ideas of momentum and adaptive learning rates, maintaining both a moving average of the gradients and their squares.
- While there is no one-size-fits-all solution, Adam is often a good default choice for training neural networks.
- But choosing the right optimiser and tuning its hyperparameters can depend on a range of different factors.

# Example: Adam Optimiser

```python
params = np.zeros(2)  # initial weights (betas)
lr = 0.01 # learning rate
beta1, beta2, eps_adam = 0.85, 0.99, 1e-8 # Adam hyperparameters
batch_size = 50
num_epochs = 1000

m = np.zeros(2)  # first moment vector
v = np.zeros(2)  # second moment vector

for epoch in range(num_epochs):
    indices = np.random.choice(len(y), size=batch_size, replace=False)
    X_batch, y_batch = X[indices], y[indices]

    y_pred = X_batch @ params
    residuals = y_batch - y_pred
    gradient_params = -2 * X_batch.T @ residuals / batch_size

    m = beta1 * m + (1 - beta1) * gradient_params # first moment update
    v = beta2 * v + (1 - beta2) * (gradient_params ** 2) # second moment update

    m_params_hat = m / (1 - beta1 ** (epoch + 1)) # bias-corrected first moment
    v_params_hat = v / (1 - beta2 ** (epoch + 1)) # bias-corrected second moment

    params = params - lr * m_params_hat / (np.sqrt(v_params_hat) + eps_adam) # parameter update

```

```
array([ 1.52080896, -2.04351973])
```

# Deep Learning Libraries

# Deep Learning Architecture

- In practice, we would rarely implement neural networks from scratch.
- Instead we would rely on a library with high-level API that abstracts away the details of the underlying computations.
- In addition to providing activation and loss functions, optimisers, etc., these libraries also make parallelisation and hardware (GPU) acceleration easier.
- Some popular deep learning libraries include:
  - **PyTorch**: Implemented in Python with C++ backend, developed by Meta, and widely used in research and industry.
  - **TensorFlow**: Developed by Google Brain for internal research and production, later open-sourced.
  - **Keras**: High-level API that can run on top of TensorFlow, PyTorch or JAX with more user-friendly interface.

# PyTorch

- `PyTorch` is one of the most popular deep learning libraries, widely used in both research and industry.
- The central computational abstraction in PyTorch is the **tensor**, which is a multi-dimensional array (similar to a NumPy array).
- Two key high-level features of PyTorch are:
  - Multi-dimensional arrays that has been optimised for GPU processing (tensors).
  - Automatic differentiation engine that computes gradients for tensor operations, enabling backpropagation (autograd).
- PyTorch can be used for building neural network models, but can also be used for simpler models that require efficient GPU computation.

# Example: Linear Regression in PyTorch

- Let's start by re-implementing our linear regression with SGD example using PyTorch instead of NumPy:

```
import torch

rng = torch.Generator().manual_seed(123)
X = torch.randn(1000, 2, generator=rng)
y = 1.5 * X[:, 0] - 2.0 * X[:, 1] + torch.randn(1000, generator=rng)
```

```
params = torch.zeros(2, requires_grad=True)  # initial weights (betas)
lr = 0.01 # learning rate
batch_size = 50
num_epochs = 1000

for epoch in range(num_epochs):
    indices = torch.randperm(len(y))[:batch_size]
    X_batch, y_batch = X[indices], y[indices]

    y_pred = X_batch @ params
    residuals = y_batch - y_pred
    gradient_params = -2 * X_batch.T @ residuals / batch_size
    params = params - lr * gradient_params

params
```

```
tensor([ 1.5580, -1.9553], grad_fn=<SubBackward0>)
```

# Example: Linear Regression in PyTorch

- Instead of manually assembling different model components, we can rely on the API built into PyTorch:

```
dataset = torch.utils.data.TensorDataset(X, y)
dataloader = torch.utils.data.DataLoader(dataset, batch_size=50, shuffle=True)

model = torch.nn.Linear(in_features=2, out_features=1, bias=False)  # linear model with 2 inputs and 1 output
criterion = torch.nn.MSELoss()  # MSE loss
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)  # SGD optimizer
num_epochs = 1000

for epoch in range(num_epochs):
    for X_batch, y_batch in dataloader:
        optimizer.zero_grad()  # reset gradients
        y_pred = model(X_batch).squeeze()  # forward pass
        loss = criterion(y_pred, y_batch)  # compute loss
        loss.backward()  # backpropagation
        optimizer.step()  # update parameters

model.weight.data
```

```
tensor([[ 1.5419, -1.9387]])
```

# Example: XOR in PyTorch

```
X = torch.tensor([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=torch.float32)
y = torch.tensor([0, 1, 1, 0], dtype=torch.float32)

dataset = torch.utils.data.TensorDataset(X, y)
dataloader = torch.utils.data.DataLoader(dataset, batch_size=4, shuffle=True)

model = torch.nn.Sequential(
    torch.nn.Linear(in_features=2, out_features=2),  # hidden layer with 2 neurons
    torch.nn.ReLU(),        # ReLU activation
    torch.nn.Linear(in_features=2, out_features=1),  # output layer with 1 neuron
    torch.nn.Sigmoid()      # sigmoid activation for binary classification
)
criterion = torch.nn.BCELoss()  # binary cross-entropy loss (NLL for binary classification)
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)
num_epochs = 1000

for epoch in range(num_epochs):
    for X_batch, y_batch in dataloader:
        optimizer.zero_grad()
        y_pred = model(X_batch).squeeze()
        loss = criterion(y_pred, y_batch)
        loss.backward()
        optimizer.step()

model(X).squeeze() # squeeze() is similar to drop=TRUE in R to remove extra dimensions
```

```
tensor([0.0226, 0.9204, 0.9204, 0.0414], grad_fn=<SqueezeBackward0>)
```

# Neural Networks for Text

# Statistical Problem

- In statistical terms we are modelling high-dimenional discrete distribution.
- For that we are using an autoregressive function that iteratively predicts the next word given the previous words:

$$P(w_t | w_{t-1}, w_{t-2}, \ldots, w_1)$$

  where the input features are the vector representations of the previous words and the output is a probability distribution over the vocabulary for the next word.
- The model can be trained to maximise the likelihood of the observed data (i.e., the next word in the sequence) given the previous words.

# Neural Language Models

- Research on applying neural networks to text has been going for a few decades.
- Bengio et al. (2003) proposed a feedforward neural network language model that learns word embeddings and can be used for next-word prediction.

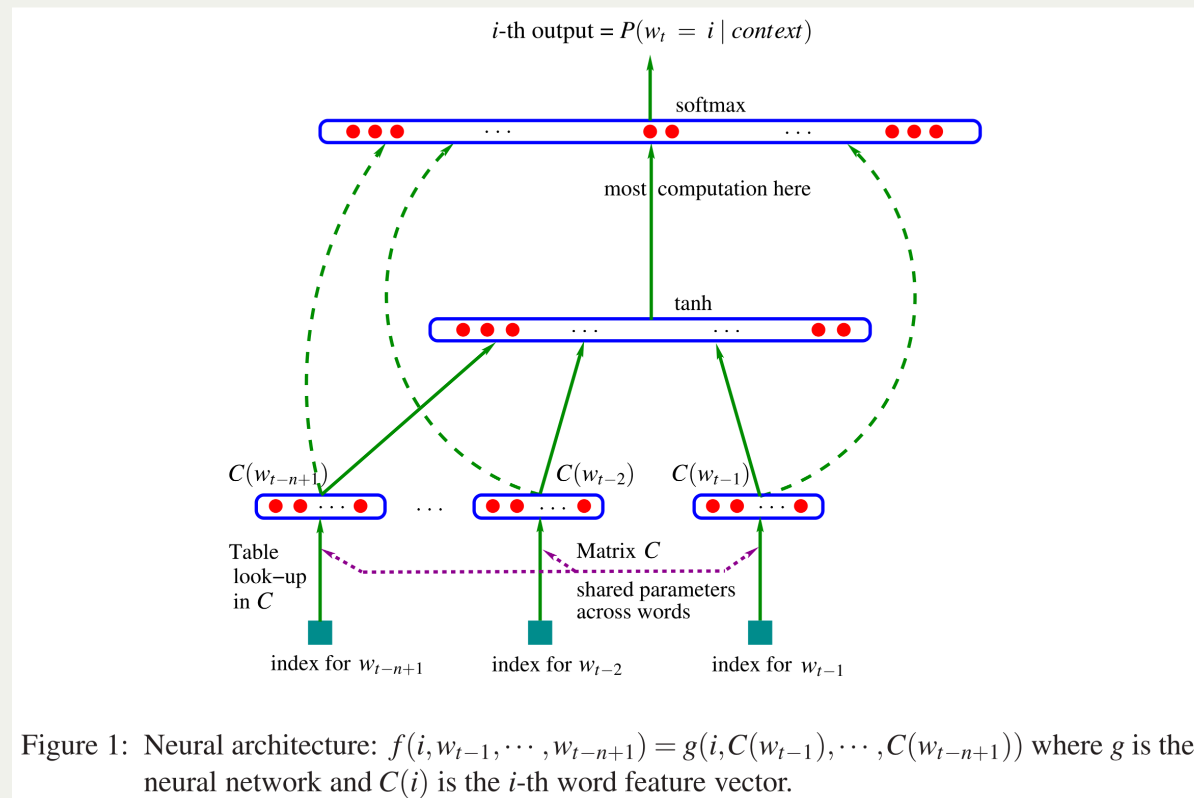



Figure 1: Neural architecture: $f(i, w_{t-1}, \cdots, w_{t-n+1}) = g(i, C(w_{t-1}), \cdots, C(w_{t-n+1}))$ where $g$ is the neural network and $C(i)$ is the $i$-th word feature vector.

(Bengio et al., 2003)

# Specialised Neural Networks

- In a typical feedforward neural network, we calculate all parameters of the model separately.
- However, this is impractical for complex real-world data, such as images, audio or text, where the number of parameters can be enormous.
- We can also surmise that there are certain regularities in the data that we can exploit (adjacent pixels in an image, adjacent words in a sentence).
- This has led to the development of specialised neural network architectures that are designed to capture these regularities, such as:
  - **Convolutional Neural Networks (CNNs)** for images.
  - **Recurrent Neural Networks (RNNs)** for sequential data (e.g., text).
  - **Transformers** for sequential data with long-range dependencies (e.g., text).

# Recurrent Neural Networks

- Recurrent neural networks (RNNs) are designed to handle sequential data by maintaining a hidden state that captures information about previous inputs in the sequence.
- They have been widely used in NLP tasks such as machine translation, image captioning, and text generation.

# Mikolov, Yih & Zweig (2013)

- The recurent neural network language model proposed by Mikolov, Yih & Zweig (2013) has the following architecture:

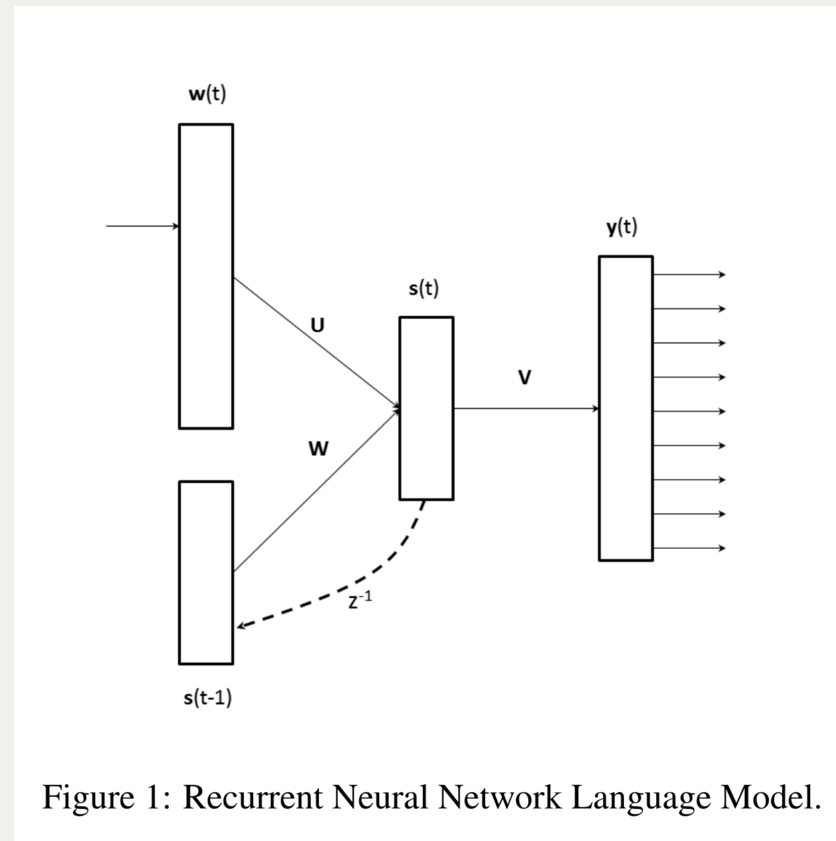



Figure 1: Recurrent Neural Network Language Model.

(Mikolov, Yih & Zweig, 2013)

where the input vector $\mathbf{w}(t)$ represents the word at time $t$ andthe output layer

$$\mathbf{y}(t)$$

is a probability distribution over the vocabulary for the next word.

# Skip-gram Model Architecture

- In Mikolov et al. (2013) the authors also propose a simpler architecture for learning word embeddings, called the **skip-gram model**, that has the following architecture:

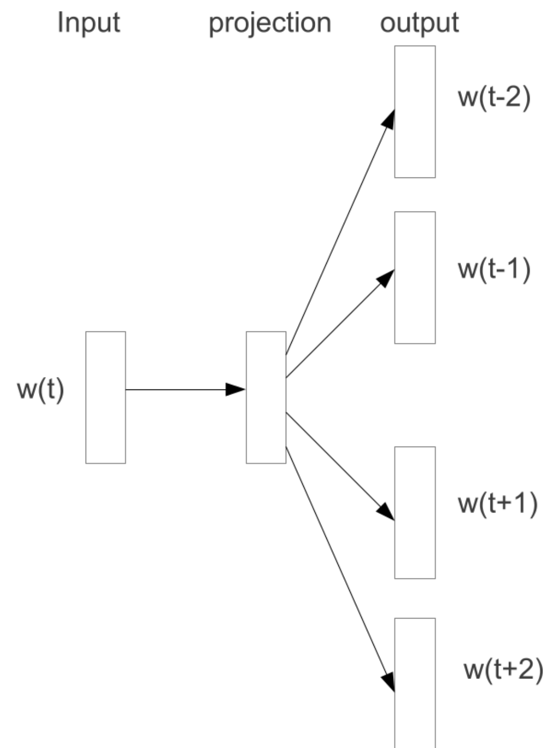



Figure 1: The Skip-gram model architecture. The training objective is to learn word vector representations that are good at predicting the nearby words.

(Mikolov et al., 2013)

# Next

- Tutorial: Neural networks for text
- Next Week: Large language models