An introduction to deep learning with the R interface to Keras

Jordan Richards[1]
(based on material by Christopher K. Wikle[2], Dan Pagendam[3] and Jarrett J. Barber[4])

[1]King Abdullah University of Science and Technology (KAUST)
([2]University of Missouri, [3]CSIRO Data61, Brisbane, Australia, [4]Northern Arizona University)

Shameless self-promotion

Deep extreme quantile regression:

- **R., J.**, Huser, R. (2022+). Regression modelling of spatiotemporal extreme U.S. wildfires via partially-interpretable neural networks.
- **R., J.**, Huser, R., Bevacqua, E., Zscheischler, J. (2023). Insights into the drivers and spatio-temporal trends of extreme Mediterranean wildfires with statistical deep-learning. Artificial Intelligence for the Earth Systems, 2(4):e220095.
- Cisneros, D., **R., J.**, Dahal, A., Lombardo, L., Huser, R. (2023+). Deep graphical regression for jointly moderate and extreme Australian wildfires.
- **R., J.** (2023+). pinnEV: R package.
- **R., J.**, Huser, R. (2023+). Extreme Quantile Regression with Deep Learning. In Handbook on Statistics of Extremes.

Shameless self-promotion 2

Deep inference:

- **R., J.**, Sainsbury-Dale, M., Zammit-Mangion, A., Huser, R. (2023+). Neural Bayes estimators for censored inference with peaks-over-threshold models.
- Sainsbury-Dale, M., **R., J.**, Zammit-Mangion, A., Huser, R. (2023+). Neural Bayes estimators for irregular spatial data using graph neural networks.
- **R., J.,**, Alotaibi, N., Cisneros, D., Gong, Y., Guerrero, M. B., Redondo, P., Shao, X. (2023+). Modern extreme value statistics for Utopian extremes.
- **R., J.** CensoredNeuralEstimators: Julia code to complement NeuralEstimators Julia/R package.
- Watch this space...

Shameless self-promotion 3

Deep dependence:

- Shao, X., **R., J.**, Huser, R. (2023+). Deep compositional models for nonstationary extremal dependence.
- DeepSpat: R package.
- Watch this space...

Outline

1. Background
2. Introduction to deep learning
   - What is deep learning?
   - Statistics vs deep learning
   - Basics of neural networks
   - Backpropogation
3. Implementation and regularisation
4. Convolutional neural networks
5. Recurrent neural networks
6. Graph neural networks

What is Keras?

- Keras is a **high-level API** for fast deep learning developed by **Google** and written primarily in **Python**, released in 2015
- Whilst it used to support a number of different **back-ends** (Theano, MILA; CNTK, Microsoft) it now solely runs on top of **Tensorflow**
- Tensorflow is a free open-source machine learning (not just DL) library written in Python, C++ and CUDA (for **GPUs**) that does all of the lower-level computations for Keras
- Keras is the **most popularly applied deep learning software** due to its simple yet powerful framework, followed closely by Facebook's PyTorch. This can also be used in R (see `https://www.rstudio.com/blog/torch/`)

Short course introduction

- Many of the ideas in deep learning (DL) have **analogues in statistics** (loss functions, gradient-based optimization, regularisation, convolutions); once you are familiar with the jargon and basic ideas, DL is fairly straightforward.
- We give a brief overview of the **methodological** and **implementation** for deep models.
- We give some examples using R and Keras.
- Specific models:
  - Feed Forward Neural Networks (FNNs; **Regression** and Classification)
  - Convolutional Neural Networks (CNNs; Image Classification)
  - Recurrent Neural Networks (RNNs; Temporal Dependence, **forecasting**)
  - Graph Neural Networks (GNNs; Node-level regression, graph-level classification)

Installation and extra

- Keras installation instructions available in *installation.html*, which should work for Windows 10, Mac OS and Ubuntu. Alternative in *installation.R* or use GoogleCollab.

- There's an html and R markdown (*keras_build*) available on my github, which gives some further details on how to build Keras models (from scratch) for non-Gaussian regression. Check there for extra details...
  ... and follow my updates while you're at it

Some suggested reading

For deep learning with R:

- Chollet, F. with Allaire, J. J. (2018). Deep learning with R.
- Any of the multitude of Keras for R blogs, see e.g., blogs.rstudio.com, r-bloggers.com, towardsdatascience, analaticsvidyha. (Even those that give code written in Python as it's very easy to translate!)
- There's some really good `https://www.coursera.org/`coursera courses for deep learning, particularly with Python

Deep learning

What is deep learning?[1]

[1] Copyright belongs to https://flatironschool.com/blog/deep-learning-vs-machine-learning/

What is deep learning?

- Machine learning is a subset of statistics and computer science
  - A set of rules (algorithm) or a model is trained, given inputs, to produce output (i.e., classification, regression)
  - meaningfully transform data to learn useful representations
- Deep learning is a subset of machine learning layered or hierarchical or multistage way to learn data representations (features)

From Chollet and Allaire (2018)

Deep learning vs Statistics

- Deep learning is tasked with estimating some quantity related to an output $\mathbf{Y}$, using inputs $\mathbf{X}$
- We can think of this as making inference about some parameter $\hat{\mathbf{Y}}$, conditional on $\mathbf{X}$, by estimating some function $f : \mathbf{X} \rightarrow \hat{\mathbf{Y}}$ - this is just regression
- Specification of $\hat{\mathbf{Y}}$ is done via some loss function and determines the type of problem we are dealing with
  - For classification (logistic/multinomial regression), $\hat{\mathbf{Y}}$ may be a label or class probability
  - For prediction (mean regression), $\hat{\mathbf{Y}}$ is $\mathbf{E}[\mathbf{Y}|\mathbf{X}]$
- The main difference between statistical, and deep, learning is the complexity of f. Statistics wants f simple and easily interpretable, e.g, linear or additive, for explainability; DL wants the most complex f that best optimises the loss

A simple example: non-linear Gaussian regression

- Consider a standard statistical problem: non-linear normal regression
- We will see that largely the **same methodology** is used to fit such models in statistics as in deep learning.
- Say we want to fit a parametric model to these data:

A simple example: non-linear Gaussian regression

A standard expectation regression model is of the form

$$Y_i = f\left(\mathbf{X}_i; \boldsymbol{\theta}\right) + \eta_i, \quad i = 1, \ldots, n$$

where $f\left(\mathbf{X}_i; \boldsymbol{\theta}\right)$ is a **non-linear parametric function** such that $\mathrm{E}\left(Y_i | \mathbf{X}_i = \mathbf{x}_i\right) = f\left(\mathbf{x}_i; \boldsymbol{\theta}\right)$. The function f could be, e.g., polynomial, exponential, etc.

- $\mathbf{x}_i$ is a $p \times 1$ vector of observed predictors
- $\boldsymbol{\theta}$ is a $q \times 1$ vector of parameters.
- η_i are i.i.d white noise.

Loss function

To fit our model, we minimize some loss function $Q(\hat{\mathbf{Y}}, \mathbf{Y})$, where $\hat{\mathbf{Y}}$ is some model output (here the conditional expectation, with $\hat{Y}_i = f(\mathbf{X}_i)$). For expectation regression, we typically select **squared-error loss** (e.g., residual sums of squares)

$$Q(f(\mathbf{x}); \mathbf{y}; \boldsymbol{\theta}) = \sum_{i=1}^{n} \{y_i - f(\mathbf{x}_i; \boldsymbol{\theta})\}^2,$$

and seek:

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \{Q(f(\mathbf{x}); \mathbf{y}; \boldsymbol{\theta})\}.$$

Note that this is equivalent to fitting a conditional Gaussian dist. (with homogeneous s.d.) via MLE. We may choose different Q depending on how $\hat{Y}$ relates to Y, e.g., classification, MAE, statistical parameters.

Estimation

We estimate $\hat{\boldsymbol{\theta}}$ via **numerical minimisation** of Q based on some iterative gradient-based algorithm of the form

$$\hat{\boldsymbol{\theta}}^{(j+1)} = \hat{\boldsymbol{\theta}}^{(j)} - \epsilon_j \, \mathrm{A}_j \frac{\partial Q\left(\hat{\boldsymbol{\theta}}^{(j)}\right)}{\partial \boldsymbol{\theta}}$$

where

- A_j is some positive definite matrix,
- $\frac{\partial Q\left(\hat{\boldsymbol{\theta}}^{(j)}\right)}{\partial \boldsymbol{\theta}}$ (i.e., $\nabla_\theta Q\left(\hat{\boldsymbol{\theta}}^{(j)}\right)$) is the gradient of the objective function $Q(\boldsymbol{\theta})$ evaluated at $\hat{\boldsymbol{\theta}}$,
- ϵ_j is the "learning rate".

Optimisation

$$\hat{\boldsymbol{\theta}}^{(j+1)} = \hat{\boldsymbol{\theta}}^{(j)} - \epsilon_j \, \mathrm{A}_j \frac{\partial Q\left(\hat{\boldsymbol{\theta}}^{(j)}\right)}{\partial \boldsymbol{\theta}},$$

- **Gradient Descent**: $A_j \equiv I_p$
- Gauss-Newton: $A_j \equiv \left[F\left(\hat{\boldsymbol{\theta}}^{(j)}\right)' \mathrm{F}\left(\hat{\boldsymbol{\theta}}^{(j)}\right)\right]^{-1}$, where $F(\theta) \equiv \left[\frac{\partial f(x_i;\theta)}{\partial \theta_j}\right]$, the $n \times p$ Jacobian matrix
- Levenberg-Marquardt: $A_j \equiv \left[F\left(\hat{\boldsymbol{\theta}}^{(j)}\right)' \mathrm{F}\left(\hat{\boldsymbol{\theta}}^{(j)}\right) + \tau\right]^{-1}$ (Ridge penalty; L_2 regularization)
- Newton-Raphson: $\mathrm{A}_j \equiv \left[\frac{\partial^2 Q\left(\hat{\boldsymbol{\theta}}^{(j)}\right)}{\partial \boldsymbol{\theta} \partial \boldsymbol{\theta}'}\right]^{-1}$ (the $p \times p$ Hessian matrix)

Optimisation

- In general, optimization in machine learning algorithms usually consider a loss function that consists of a sum or average over the **entire training data** (e.g., all n observations).
- If n is large, computation of the gradient can be slow. In addition, there may be **redundant data** in the training sample.

Optimisation: SGD

- One way to mitigate the aforementioned issues is to consider minimizing the **expected loss**. Estimation of the expected loss can be achieved by averaging the loss over small samples (**batches** or **minibatches**) of the training data (assuming the inputs x_i are independently drawn from the same underlying distribution; not always reliable).
- We refer to this as **stochastic** gradient descent, as the expectation of the gradient is estimated by **randomly** sampling a small number of samples of the training set and taking the average- this turns out to be a very effective solution to this problem.
- Typically, smaller batch sizes offer more regularization and more flexibility (generalization) outside the training sample.

SGD algorithm

Basic Stochastic Gradient Descent (SGD) Algorithm (Goodfellow et al. 2016, Chap. 8)

Require: Learning rate ϵ
Require: Initial parameters $\boldsymbol{\theta}^{(j)}$
  while Stopping criterion not met **do**
    Sample a minibatch of m inputs $\{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$
      and targets $\{\mathbf{y}_1, \ldots, \mathbf{y}_m\}$
    Set $g = 0$
    for $i = 1$ to m **do**
      Compute gradient estimate: $g \leftarrow g + \frac{1}{m} \nabla_{\boldsymbol{\theta}} Q\left(f\left(\mathbf{x}_i; \boldsymbol{\theta}\right), \mathbf{y}_i\right)$
    end for
    Apply update: $\boldsymbol{\theta}^{(j+1)} \leftarrow \boldsymbol{\theta}^{(j)} - \epsilon \mathrm{g}$
  end while

SGD (cont.)

- The learning rates are **quite important**. Large rates will cause the optimization to diverge; small rates may cause convergence to local minima. Algorithms exist for adaptive tuning of ϵ, most notably RMSprop[2] and ADAM[3].
- We may sometimes want to regularise the parameters in our model. This can be done by adding L_2 (ridge) or L_1 (lasso) penalties to the loss!
- The framework is exactly the same for Deep learning, except now $f(\mathbf{X}_i; \boldsymbol{\theta})$ is a multi-layered neural network!

[2] Unpublished Coursera course by Geoff Hinton
`https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`

[3] `https://arxiv.org/abs/1412.6980` with over 111000 citations!

Basics of neural networks

- A neural network is an algorithm designed to mimic the **human brain**.
- It can be constructed as a **directed graph** with a single input and a single output, with a dense network of **interconnected nodes** in-between
- Data is input into the graph and information is extracted at each node, a.k.a **perceptron**. An output is provided at the end of the network.
- Similarly brain activity occurs when a stimuli enters the system, information is based through the network via **neurons** that extra relevant information and this information passes to some area of the brain that forces a response (output)

Basics of neural networks (cont.)

Nodes in a neural network are arranged in layers:

- A single node may be connected to several nodes in the layer above, from which it **receives information**, and to the layer below, which it **feeds information**
- Most neural networks are **"feed forward"** - information is passed forward, layer-to-layer, in one direction only (but we also consider RNN)
- Different information (features) is extracted at each layer in a **hierarchical** fashion, i.e., the most important aspects first
- The "deep" in deep learning refers to the depth of layers in a neural network as well as its hierarchical nature
- Each layer can be considered as a non-linear regression of inputs from the previous layer; a neural network is simply a composition of non-linear regressions

General structure

- A **single input layer**. Each input would be a single predictor variable in the form of a scalar value, a sequence or an image (or even a sequence of images!).
- **A single output layer**. For prediction or classification problems, we would have a single node here.
- A collection of **hidden layers** of variable **widths**. Preferably more than one!
- Calculations are completed at each node in the hidden layer. These are parameterised by **weights and biases**.
- In Keras, calculations within a layer are of a certain "type". We will consider standard (**vanilla**), convolution and recurrent layers.

We refer to the structure of a neural network as its **architecture**.

Single layer neural network

- Let's first consider a simple feed-forward NN with a single hidden layer, a.k.a, a **single layer perceptron**
- Takes input p-dimensional input $\mathbf{x}$, k-dimensional output $\hat{\mathbf{y}}$ (although $k = 1$ for most problems) and single hidden layer $\mathbf{z}$ (hence "**shallow**")

FIGURE 11.2. *Schematic of a single hidden layer, feed-forward neural network.*

[Hastie et al. 2009, Fig. 11.2.]

Single layer FNN: hidden layer

A hidden layer in a perceptron is implemented using `layer_dense` in `Keras`.

Hidden Layer:

$$z_j = g\left(\alpha_{j0} + \sum_{i=1}^{p} \alpha_{ji} x_i\right), \quad j = 1, \ldots, J,$$

where z_j is the hidden variable, $\alpha_{j0} \in \mathbb{R}$ is bias, $\{\alpha_{ji}\} \in \mathbb{R}$ are the weight parameters, and $g(\cdot)$ is the activation function.

It simplifies notation to assume $x_0 = 1$ and then

$$z_j = g\left(\sum_{i=0}^{p} \alpha_{ji} x_i\right), \quad j = 1, \ldots, J$$

Activation functions[4]

The activation function defines the **output of the node**. There's a range of different functions that you can use (see `https://keras.rstudio.com/reference/activation_relu.html`), but typically they must satisfy two properties:

- **Nonlinear**! If all nodes had linear activation, this would just be a linear regression model. A two-layered NN with non-linear activations is a **universal function approximator**.
- **"Continuously" differentiable**. Needed for gradient-based optimisation.

You can think of the activation function as an analogue to the link function in regression.

[4] `https://en.wikipedia.org/wiki/Activation_function`

ReLu activation

An activation function that's popularly applied is the **rectified linear unit**:

- Older neural networks relied on sigmoid or tanh activation functions
- These suffered from the *vanishing gradient problem*[5], which made it difficult to efficiently train deep networks
- The ReLU replaced these. It has the form $g(x) = \max\{x, 0\}$.

- The neuron "activates" only if **enough information passes through** (think pain receptors)
- One drawback: not differentiable at zero

[5] The networks we will be considering will not be overly deep, so this problem does not occur. For details, see `https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/`

Single layer FNN: output layer

Output Layer:

$$\hat{y}_k = h\left(\beta_{k0} + \sum_{j=1}^{J} \beta_{kj} z_j\right), \quad k = 1, \ldots, K,$$

where $\hat{y}_k$ is the output variable and $h(\cdot)$ is another activation function, which must be carefully selected to enforce range constraints on the output $\hat{\mathbf{y}}$. We may take the identity function for mean/median regression or a sigmoid or softmax function for classification.
As with the hidden layer, we can write

$$\hat{y}_k = h\left(\sum_{j=0}^{J} \beta_{kj} z_j\right), \quad k = 1, \ldots, K.$$

Singe layer FNN (cont.)

In summary, the single layer feed-forward NN can be written:

$$\hat{y}_k(\mathrm{x}; \boldsymbol{\beta}, \boldsymbol{\alpha}) = h\left(\sum_{j=0}^{J} \beta_{kj} g\left(\sum_{i=0}^{p} \alpha_{ji} x_i\right)\right).$$

As with traditional statistical regression, we estimate the parameters (or "train the network") by choosing an objective function $Q(\hat{\mathbf{y}}, \mathbf{y}; \boldsymbol{\alpha}, \boldsymbol{\beta})$ and then use stochastic gradient descent to obtain parameter estimates of the weight/bias vectors, $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$.
Note that this formulation illustrates that an NN is just a **composition of non-linear activations of convex sums**.

Loss functions

Whilst the NN architecture can be standard for any task, the loss function Q must be **very specific**! Examples include:

- MSE: $(1/n)\sum_{l=1}^{n}(y_l - \hat{y}_l)^2$ (mean prediction)
- MAE: $(1/n)\sum_{l=1}^{n}|y_l - \hat{y}_l|$ (median prediction)
- binary cross-entropy (if $\hat{y}_l \in (0,1)$ and $y_l = 0$ or 1): $-(1/n)\sum_{l=1}^{n}(y_l \log \hat{y}_l + (1-y_l)\log(1-\hat{y}_l))$. Two-class classification, but can be extended to more than two classes.
- **negative log-likelihood**, where $\hat{y}$ are model parameters

Backpropagation

In order to use SGD, we require the negative log-likelihood of Q w.r.t the weight/biases. This may seem like a fairly daunting task, particularly if we have many weights or additional layers. However, the neural net community uses an algorithm called "**backpropagation**" to evaluate the gradients very quickly; this is nothing more than iterative application of the chain rule, which is particularity easy due to the hierarchical nature of the model.

$$\frac{\partial Q}{\partial \alpha_{ji}} = \frac{\partial Q}{\partial \hat{y}_k}\frac{\partial \hat{y}_k}{\partial z_j}\frac{\partial z_j}{\partial \alpha_{ji}}, \quad j = 1, \ldots, J; i = 0, \ldots, p$$
$$\frac{\partial Q}{\partial \beta_{kj}} = \frac{\partial Q}{\partial \hat{y}_k}\frac{\partial \hat{y}_k}{\partial \beta_{kj}}, \quad k = 1, \ldots, K; j = 0, \ldots, J.$$

Illustrating back propagation

Let's take a simple illustration. Consider a fairly simple case of nonlinear regression with a single output, $h(\cdot)$ the identity function, $g(\cdot)$ the sigmoid function, and squared error loss.

$$\hat{y} = \sum_{j=0}^{J} \beta_j z_j, \tag{1}$$

$$z_j = g\left(\sum_{i=0}^{p} \alpha_{ji} x_i\right). \tag{2}$$

Squared-error objective function:

$$Q(\hat{y}, y; \boldsymbol{\alpha}, \boldsymbol{\beta}) = \frac{1}{2}(y - \hat{y})^2.$$

Illustrating back propagation

The gradients are given by:

$$\frac{\partial Q}{\partial \beta_{kj}} = \frac{\partial Q}{\partial \hat{y}_k}\frac{\partial \hat{y}_k}{\partial \beta_{kj}} = -\left(-\hat{y}\right) z_j \tag{3}$$

$$\begin{aligned}\frac{\partial Q}{\partial \alpha_{ji}} &= \frac{\partial Q}{\partial \hat{y}_k}\frac{\partial \hat{y}_k}{\partial z_j}\frac{\partial z_j}{\partial \alpha_{ji}} \\ &= \left\{-\left(y-\hat{y}\right)\right\}\left\{\beta_j\right\}\left\{z_j\left(1-z_j\right)x_i\right\}\end{aligned} \tag{4}$$

Note: $\partial g(ab)/\partial a = g(a)(1-g(a))b$ when $g(\cdot)$ is a sigmoidal function. The gradients extend **as expected** when more hidden layers are introduced, e.g., a two layered model will have an extra set of weights, and three equalities with a product of two, three and four terms.

Backpropogation (cont.)

The gradient vectors fed in SGD and **updates are done in a two-pass algorithm**.

- In the **forward pass**, the current weights values are fixed and z_j and $\hat{y}$ are calculated using (1) and (2)
- In the **backward pass**, the gradients are then calculated from (3) and (4); then, the parameters are updated using the SGD updates shown earlier.

An important feature of backpropagation algorithms is that each hidden unit passes and receives information only to and from units that share a connection - i.e., it is a "local" algorithm. This facilitates computation in a parallel computing environment, which is important for large datasets.

Automatic differentiation

- Note that you **never have to calculate derivatives** to implement backpropagation.
- Deep learning software packages all include some form of automatic differentiation that can symbolically/algorithmically work out the derivatives for you. Derivatives for standard loss and activations will be hardcoded explicitly.
- As long as we construct **differentiable loss and activation functions**, backpropagation can be performed very efficiently! When writing custom functions for `Keras`, we **must use the** `tensorflow` or `Keras` backend functions; these have **known derivatives** that `tensorflow` can easily call.

Deep neural networks

Putting the **deep** in **deep** learning:

- Deep neural networks generalise the previously seen single layer network to **multiple layers**
- The outputs from one layer become the input of the next. We consider the number of units in each layer as the **width** and the number of layers as the **depth** of the network. Having both width and depth provides a **very flexible learning environment**.
- The biggest challenge with implementing deep models is that they are often **severely over-parameterized** and easily over-fit
- With many parameters, we may also experience **vanishing** or **exploding** gradients; theses issues can be mitigated with data pre-processing and weight initialisation

Typical structure[6]

[6] Copyright belongs to https://www.ibm.com/cloud/blog/ai-vs-machine-learning-vs-deep-learning-vs-neural-networks

Deep FFNN: 2-layered example

An MLP with two hidden layers and one output can be written:

$$z_j^{(1)} = g^{(1)}\left(\sum_{i=0}^{p} \alpha_{ji}^{(1)} x_i\right), \quad j = 1, \ldots, J_1,$$

$$z_{j'}^{(2)} = g^{(2)}\left(\sum_{j=1}^{J_1} \alpha_{j'j}^{(2)} z_j^{(1)}\right), \quad j' = 1, \ldots, J_2,$$

$$\hat{y} = h\left(\sum_{j'=0}^{J_2} \beta_{j'} z_{j'}^{(2)}\right).$$

Training follows with backpropagation in the same way as for the one-hidden layer model; the gradient vectors generalise as expected.

Deep FFNN: Matrix notation

An MLP with L hidden layers can be written:

$$\mathbf{z}^{(1)} = \mathbf{g}^{(1)}\left(\mathbf{W}^{(0)}\mathbf{x}\right),$$
$$\mathbf{z}^{(l)} = \mathbf{g}^{(l)}\left(\mathbf{W}^{(l)}\mathbf{z}^{(l)}\right), \quad l = 1, \ldots, L$$
$$\hat{\mathbf{y}} = \mathbf{h}\left(\mathbf{W}^{(L)}\mathbf{z}^{(L)}\right).$$

where $\mathbf{z}^{(l)} = (z_1^{(l)}, \ldots, z_{J_l}^{(l)})$, $\mathbf{W}^{(0)} \in \mathbb{R}^{J_1 \times p}$, $\mathbf{W}^{(l)} \in \mathbb{R}^{J_l \times J_{l-1}}$.
Functions $\mathbf{g}^{(l)}$ and $\mathbf{h}$ are applied component-wise. Note that $\hat{\mathbf{y}}$ is a vector or a scalar.

Full specification for conditional density estimation

Let's say $Y \mid \mathbf{X} = \mathbf{x} \sim \mathcal{F}(\hat{\mathbf{y}})$. Maybe $\mathcal{F}(\hat{\mathbf{y}}) \sim \Gamma(\hat{y}_1, \hat{y}_2)$ with shape $\hat{y}_1 > 0$ and scale $\hat{y}_2 > 0$.
Then we can build a deep Gamma regression model using an L-layered MLP. So

$$\mathbf{z}^{(1)} = \mathbf{g}^{(1)}\left(\mathbf{W}^{(0)}\mathbf{x}\right),$$

$$\mathbf{z}^{(l)} = \mathbf{g}^{(l)}\left(\mathbf{W}^{(l)}\mathbf{z}^{(l)}\right), \quad l = 1, \ldots, L$$

$$\hat{\mathbf{y}} = \mathbf{h}\left(\mathbf{W}^{(L)}\mathbf{z}^{(L)}\right).$$

What should $\mathbf{h}$ be?
How do we train this model?

Full specification for conditional density estimation

Let's say $Y \mid \mathbf{X} = \mathbf{x} \sim \text{GEV}(\hat{y}_1, \hat{y}_2, \hat{y}_3)$ with location $\hat{y}_1 \in \mathbb{R}$, scale $\hat{y}_2 > 0$ and shape $\hat{y}_3 \in (0, 1)$.
Then we can build a deep GEV regression model using an L-layered MLP. So

$$\mathbf{z}^{(1)} = \mathbf{g}^{(1)}\left(\mathbf{W}^{(0)}\mathbf{x}\right),$$

$$\mathbf{z}^{(l)} = \mathbf{g}^{(l)}\left(\mathbf{W}^{(l)}\mathbf{z}^{(l)}\right), \quad l = 1, \ldots, L$$

$$\hat{\mathbf{y}} = \mathbf{h}\left(\mathbf{W}^{(L)}\mathbf{z}^{(L)}\right).$$

What should $\mathbf{h}$ be?
The only other difference in the MLP specification is that now $\hat{\mathbf{y}}$ has length 3.

Some things to consider when implementing large models to avoid numerical instabilities in training or overfitting

Training/validation/testing

Due to their large number of parameters, neural networks are prone to overfitting. It's necessary to use **validation/testing** to assess any overfitting.

- First, we subset the data into training, validation and testing. Opinions differ, but the standard is 80/10/10
- Each set has a different purpose:
  - The model is **trained** on the training data. This is used to optimise the weights, biases and other hyper-parameters
  - We then **validate** the model by getting predictions for the validation set and evaluating validation loss. This will give us some insight as to whether or not overfitting is occurring. Typically the validation loss is used for model selection
  - To get a truly unbiased evaluation of the model fit and to compare amongst different models, we should **test** the model on previously unseen data

Mitigation

- Some literature uses **test** and **validation** interchangeably, e.g., in cross-validation, the validation (hold-out) set is actually a test set because the model never sees it for training
- Overfitting in neural networks is **inevitable** if a network is trained **indefinitely**.
- However, you can mitigate the risk of fitting using forms of regularisation.

Initialisation

Although the backpropagation algorithm is simple, the challenge is to implement it in a way that is practical and generalizable, particularly for large networks. Some considerations should be taken when initialising a model.

Normalisation/standardisation:

- The inputs into a neural network should be standardised so that features have the same scale. Common approaches include: subtracting the mean and dividing by the standard deviation; mapping all features to $[0, 1]$ by subtracting the max and dividing by the range. Note that both the training and test/validation data must be standardised in the exact same manner
- Batch normalisation: outputs of a hidden layer within a neural network are centered and scaled using the `layer_batch_normalization` in `Keras`

Initialisation

Weight initialization: Given the highly nonlinear nature of neural models, they can be very sensitive to the initial parameter values. Two general heuristic "principles" to consider:

- choose initial weights such that each hidden variable is operating in the linear range of its activation function, i.e., not so large or small that the derivative is initialised at zero; as this depends on both the weights and the inputs, this is why we standardize the inputs
- choose initial weights randomly (hidden units are interchangeable; one needs them to find different features, so it is important that they not have the same weights) (default in Keras)

Exception: pre-training, **pathological likelihoods**

Regularisation

In order to mitigate overfitting, consider the following regularisation techniques:

Penalties: The most common is to consider an L_2 (ridge) penalty to the objective function of the form, e.g., in our simple example, $\lambda\left(\sum_{ji}\alpha_{ji}^2+\sum_{kj}\beta_{kj}^2\right)$, where $\lambda \geq 0$ is some tuning parameter. This has the effect of adding terms $2\lambda\beta_{kj}$ and $2\lambda\alpha_{ji}$ to the gradient expressions, and shrinks the weights to zero.

Can use L_1 (Lasso) as an alternative - this provides full weight elimination, in which small parameters are set to zero - or a mix of the two.

Regularisation

Dropout: An alterative form of regularisation is **dropout**.
The idea is straightforward: randomly omit a certain (relatively small) percentage (say, γ) of the hidden units in each hidden layer for each minibatch of samples during the backpropagation training.
This helps to **break up the dependence** in the hidden units and prevents overfitting. It is easy to implement dropout - just set the activation to 0 (no information propagates through that unit, and so the algorithm doesn't have to be changed in any other way).
Implemented via `layer_dropout` in `Keras`.

Regularisation

Early stopping:
Rarely does the best generalization of a deep model occur at the local minimum obtained by the gradient descent algorithm. Thus, an important form of regularization that facilitates generalization is called early stopping, which simply means one stops the gradient descent algorithm before it has fully converged to a minima of the loss.
Neural networks are trained for a pre-fixed number of iterations (**epochs**). At which epoch to stop training can be inuited as an extra hyperparameter that must be chosen **via validation** or by trial-and-error. This is the easiest type of regularisation one can do as it doesn't require any change to the basic algorithm and can be used (and typically is) with other forms of regularization (e.g., weight decay, dropout, etc.).

Regularisation

Weight sharing and parsimonity:

Reduce the number of trainable weights!

In some types of deep models, it makes sense that weights should be similar (e.g., imagine weights connected to components of an image or in a sequence of words). One can force these weights to be identical, or can add a penalty term that favours them being similar (we will see such an example with convolutional NNs).

Using a neural network that's adapted to your task may provide you with a model that's easier and faster to train, with less estimable parameters.

Regularisation

Data augmentation

Neural networks typically require a lot of data to train! Overfitting is more prevalent with fewer training samples from which to learn, which prevents the model from being able to generalize to new data.

For some models (e.g., CNNs), it is possible to use data augmentation to **generate more training data from existing training samples** by transforming inputs to produce new input samples that are similar, but different, to the training data, e.g., rotating images.

Regularisation

Pre-training

We can often save time when training networks by using the parameters from a pre-trained model as initial values in backpropogation. This is particularly useful if you want:

- Repeated model fits, e.g., a **bootstrap analysis**
- **To train a very complex model**. Very complicated models are commercially trained on massive datasets (e.g., ImageNet, which has 1000 **categories** and 1.4 million images), with substantial computational resources. A user can take a pre-trained model and then re-train the model on their own data (potentially treating some hidden layers as fixed a priori)

Hyper-parameters

There are hyper-parameters we need to consider when building and training neural networks:

Batch-size:

As discussed, using batches of data in stochastic gradient descent is **computationally advantageous**. In deep learning, one divides the training samples by the batch size and cycles through all training data in a single **epoch**.

Smaller batch-size $\Rightarrow$ faster training, but more volatile

Larger batch-size $\Rightarrow$ slower training, but more regular convergence (particularly with **highly dependence inputs**!)

It may be preferable to have large gradient variance at the beginning of the optimization so the solution can jump out of local minima; hence we could also use a varying sample size across training.

Hyper-parameters

Architecture:
The number of layers and the number of hidden units in each layer are also hyper-parameters that need to be chosen. Other than the fact that we need enough hidden units to capture structure, and we don't want "too many" (to prevent overfitting), choosing the complexity isn't an exact art. This is very much the "art" of deep modelling.
In general, wide/shallow models are easier to overfit and narrow/deep models are easier to underfit. It is often recommended that the lower levels have more hidden units, but bottlenecks in networks can also be helpful.
We can choose the architecture through model **testing** (not validation!). I typically start with a standard architecture (across all projects) and then just iteratively test changes in complexity.

Hyper-parameters

Others:

- Learning rate decay in optimisers
- Dropout rate
- Smoothing penalty tuning parameters
- Early-stopping

Convolutional neural networks

Convolutional neural networks (CNNs)

- One of the early success stories in multi-level neural networks occurred with so-called "**convolutional neural networks** (CNNs)" applied to image classification (e.g., LeCun and colleagues in the 1990s).
- CNNs are successful because the patterns they learn are **translation invariant** and they can learn **spatial hierarchies of patterns**. Whilst densely-connected networks learn global patterns in their input feature space, convolution layers learn local patterns.
- Before we talk about convolutional networks, we review a bit about convolution, especially as it applies to images.

Convolution

Basic Mathematical Definition:

$$(f * g)(t) = \int_{\mathbb{R}^d} f(\tau)g(t-\tau)d\tau = \int_{\mathbb{R}^d} f(t-\tau)g(\tau)d\tau.$$

Discrete convolution (e.g., $\mathrm{d} = 2\mathrm{dim}$) [in practice, these are finite sums]

$$f[x,y] * g[x,y] = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} f[i,j]g[x-i, y-j]$$

So, we can think of $f[]$ as a "kernel" weight function that is applied to elements of the spatial image $g[]$. We are familiar with this concept in terms of kernel smoothing; note that in kernel smoothing, the kernels must integrate to 1 (or sum, in the case of discrete data) - this is not the case for convolution - but, we sometimes still call them **kernels** or **filters**.

Convolution: 1-d example

g[ ] – from random N(0,1)

f[ ] – normal kernel with different bandwidths

©C. K. Wikle and D. Pagendam

Convolution: 2-d example

©C. K. Wikle and D. Pagendam

Note the **edge effect** - the convolved image is smaller. This can be avoided by **padding** the original image with zeroes around the edges.

Image convolutions[7]

Different kernel weights extract different features from images.
Consider this handsome fellow:

[7] Following done using the `OpenImageR` package

Image convolutions

What happens when we apply this first kernel?

$$* \frac{1}{25} \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix} =$$

Image convolutions

And this second one?

$$* \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} =$$

Image convolutions

If we subtract the two images, we see that the kernel shifts the image 1 pixel to the left.

Now we're picking up some features!

Image convolutions

Third kernel? Sharpens.

$$* \left[\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} - \frac{1}{25} \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix} \right] =$$

Image convolutions

x-axis edge detection

$$* \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} =$$

Image convolutions

y-axis edge detection

$$* \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} =$$

Image convolutions

Laplacian kernel - all edge detection

$$* \begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix} =$$

Image convolutions

Random kernel - all weights drawn from Unif(-2,2)

Lots of different features extracted. Ideally, we want an algorithm that can learn the kernel weights, rather than pre-specifying them. We call every convolved image in a CNN a **feature map**.

Image convolutions

Tensors: In practice, input (images) are often tensors, so that there is a multivariate value at each spatial location (pixel). For example, in **RGB** color images, there are three values (associated with red, green and blue intensities) that define the overall colour of a pixel color.
If the input is RGB, a CNN considers a tensor convolution where the kernel averages over location and the variables (referred to as "**channels**").

Convolution layers[8]

A **convolution layer** acts on an image, not a vector of inputs. The image can be 1D (e.g., text), 2D or 3D (e.g., a video or 3D model)

- A convolution filter/kernel of a certain dimension is passed over the image
- The filter starts in a corner of the image. A convolution is performed, and then the filter moves one horizontal "stride". Once it reaches the edge, it moves one vertical stride down.

[8] Copyright belongs to `https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks/`

Convolution layers[9]

- After the filter has passed over the entire image, a **feature map** will be produced. The dimension of the feature map will be determined by the filter dimension and the strides
- An image can have multiple "channels", i.e., different colours, inputs, each with its own filter
- The feature maps for the different channels are added, a bias is introduced and then *a* is applied
- A layer can have **multiple filters**, similar to nodes in a densely-connected network

[9] Copyright belongs to `https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks/`

CNNs

- One of the strengths of CNNs is that it learns kernel weights, but **reduces the parameter space** by assuming that there is only one set of weights for each convolution. This ability to share weights leads to a huge dimension reduction in the parameter space (compared to a DNN).
- Then CNN considers a convolution of the image with unknown weights that are learned.
- Then the convolution step is followed by a **pooling layer** (or, "subsampling" or "downsampling"), which further reduces the dimension.

Pooling

A limitation of convolutional layers is that they **record the precise position of features in the input**. This means that small movements in the position of the feature in the input image will result in a different feature map. This can happen when applying cropping, rotation, shifting, and other minor changes to the input image.

- We can make a CNN more **robust to image translations** by using downsampling. This can be achieved by varying the stride of the convolution kernel or (more commonly) via pooling.
- The pooling layer considers a small rectangular block from the convolutional step and subsamples it in some way to produce a single output. The most common type of pooling is **max-pooling**, where we just sample the maximum value in a block (from a 2×2 window with stride 2).
- Note that this is also a form of dimension reduction.
- Implemented with `layer_max_pooling_2d`.

Max-pooling: example

With Laplacian kernel - all edge detection

$$* \begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix} =$$

Max-pooling: example

$* \; 2 \times 2$ max-pool

Max-pooling: example

$* \; 5 \times 5$ max-pool

Note that the same hidden feature map is produced even if the original image is shifted up to five pixels in any direction.

CNN architecture

The general structure of a CNN has alternating convolution layers and pooling layers, with the image/feature map dimension decreasing monotonically; the last layer is **flattened and fully connected**, as with MLPs.

From A. Gibiansky's blog

- For each convolution, multiple feature maps are created using different filters with different weights
- Bias is added to the convolved images and then they go through an activation function - typically ReLU

CNN architecture

Original LeCun CNN for MNIST handwriting:

Figure 3: Input image (left), weight vector (center), and resulting feature map (right). The feature map is obtained by scanning the input image with a single neuron that has a local receptive field, as indicated. White represents -1, black represents +1.

CNN implementation

- The critical stage of the CNN that requires training is the convolution step; the pooling layers are typically pre-specified and are not learned).
- Parameters for convolutions for any given filter are the same for the entire feature map. This parameter sharing is important as it allows the presence of a shape in any part of the feature map to be processed in the same way, regardless of its particular spatial location; i.e., it is **translation invariant**.
- Each feature in the final convolution layer is connected to each hidden state in the first fully connected layer. This layer works exactly like the layers in a deep FNN. Most of the parameters in a CNN occur in these final layers (e.g., if each of two fully connected layers has 4096 hidden units, then the connections between them will have 16 million weights!).

CNN implementation

- As before, we use **backpropagation** to train the CNN.
- The points about **regularisation** and **over-fitting** still apply!
- Two extra considerations:
  - Images can be padded to keep the dimension the same; useful if you want to output an image, rather than a vector $\hat{\mathbf{y}}$!
  - **Data augmentation**; unlike FNNs, we can add extra samples for training through random transformations of images!

Data augmentation

Original image

Augmented cats

https://towardsdatascience.com/deep-learning-3-more-on-cnns-handling-overfitting-2bd5d99abe5d

Recurrent neural networks

Recurrent neural networks

The types of FFNNs that we have covered thus far do not explicitly account for **sequential/temporal dependence** between inputs and outputs.
This gives them limited capacity for solving tasks which depend on the sequential nature of an input, e.g.,

- Document and time-series classification
- Speech recognition
- Sequence-to-sequence learning (language translation)
- Sentiment analysis
- Forecasting

Recurrent neural networks

Recurrent Neural Networks (RNNs) were originally developed in the 1980s as feedback models for sequence data. In recent years, they have become one of the **most used and successful deep learning methods**, particularly for language processing applications.
These models are closely related to multivariate state-space models for **dynamical systems**, as one might see in time series, econometrics, or spatio-temporal statistics.

Dynamical systems

A classical dynamical system can be written by

$$\mathbf{s}_t = f\left(\mathbf{s}_{t-1}; \boldsymbol{\theta}\right),$$

where $\mathbf{s}_t$ represents the state of the system at time t. This is considered "recurrent" because the state at time t refers back to the state at time $t-1$. We can "unfold" this into

$$\mathbf{s}_t = f\left(\mathbf{s}_{t-1}; \boldsymbol{\theta}\right) = f\left(f\left(\mathbf{s}_{t-2}; \boldsymbol{\theta}\right); \boldsymbol{\theta}\right) = f\left(f\left(f\left(\mathbf{s}_{t-3}; \boldsymbol{\theta}\right); \boldsymbol{\theta}\right); \boldsymbol{\theta}\right) \ldots$$

Note, the parameters θ are shared across all states.

RNNs[10]

A recurrent layer takes in as input a sequence of vectors/images. Let's say that the input is the sequence of vectors $\{\mathbf{x}_t : t = 1, \ldots, T\}$ where $\mathbf{x}_t \in \mathbb{R}^d$ for all t.

Once an input enters the layer, it becomes stuck in a **recurrence loop** - the output from the computation using the first input vector $\mathbf{x}_1$ becomes an input for the computation on $\mathbf{x}_2$ (and so on...).

[10] Copyright belongs to `https://www.ibm.com/cloud/learn/recurrent-neural-networks`

RNNs

In an RNN, as with a traditional state space model, the states are typically considered "hidden" (we denote these hidden states $\mathbf{h}_t$) and add external inputs $\mathbf{x}_t$:

$$\mathbf{h}_t = f\left(\mathbf{h}_{t-1}, \mathbf{x}_t; \boldsymbol{\theta}\right)$$

So, sequences are processed by iterating through the sequence elements and maintaining a state containing information relative to what it has seen so far via an internal feedback loop. The following image shows how this is unfolded into a computational graph structure:

©C. K. Wikle and D. Pagendam

RNNs

A single-layered in a basic ("vanilla") RNN can be written as

$$\hat{\mathbf{y}}_t = g_o\left(\mathbf{o}_t\right)$$
$$\mathbf{o}_t = \mathbf{W}_{h\hat{y}}\mathbf{h}_t$$
$$\mathbf{h}_t = g_h\left(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{xh}\mathbf{x}_t\right)$$

for activation functions $g_o(\cdot), g_h(\cdot)$, and weight matrices $\mathbf{W}_{h\hat{y}}$, $\mathbf{W}_{hh}$, and $\mathbf{W}_{xh}$ (typically, these contain bias terms as well).
Note that **equivalence with a FFNN** is given by setting all elements of $\mathbf{W}_{hh}$ to zero!

Some points...

- Weights in an RNN layer are estimated using "**backpropogation through time**" - a slight extension of backpropogation which accounts for the shared parameters across time
- RNN layers can be stacked on top of each other - we can choose for `Keras` to output the entire series of vectors in a hidden layer $(\mathbf{h}_t)_{t=1}^{T}$ or just one vector, e.g., $\mathbf{h}_T$
- These simple layers can be implemented using `layer_simple_rnn`
- Input into `Keras` is a tensor with axes $[n, T, p]$

LSTMs

- Generally simple RNNs aren't used in practice now due to the so-called vanishing/exploding problem. Diminishing time series dependence leads to exponentially smaller weights for long time series.
- Not all information from the past needs to be remembered! Some RNNs add "**gates**" that selectively allows information from the past to influence the future state. These gates retain information for later and have their own internal memory.
- One of the most famous type are **Long Short-Term Memory** networks[11], implementable by `layer_lstm`.

[11] S. Hochreiter; J. Schmidhuber (1997). "Long short-term memory". Neural Computation. 9 (8): 1735–1780.

LSTMs

Consider the basic one-layered LSTM:

$$
\begin{aligned}
\text{Output:} \quad & \mathbf{y}_t = g_o\left(\mathbf{V}\mathbf{h}_t\right) \\
\text{Hidden State:} \quad & \mathbf{h}_t = \tanh\left(\mathbf{c}_t\right) \odot \mathbf{o} \\
\text{Internal Memory:} \quad & \mathbf{c}_t = \mathbf{c}_{t-1} \odot \mathbf{f} + \mathbf{g} \odot \mathbf{i} \\
\text{Candidate Hidden State:} \quad & \mathbf{g} = \tanh\left(\mathbf{U}^g\mathbf{x}_t + \mathbf{W}^g\mathbf{h}_{t-1}\right) \\
\text{Output Gate:} \quad & \mathbf{o} = g\left(\mathbf{U}^o\mathbf{x}_t + \mathbf{W}^o\mathbf{h}_{t-1}\right) \\
\text{Forget Gate:} \quad & \mathbf{f} = g\left(\mathbf{U}^f\mathbf{x}_t + \mathbf{W}^f\mathbf{h}_{t-1}\right) \\
\text{Input Gate:} \quad & \mathbf{i} = g\left(\mathbf{U}^i\mathbf{x}_t + \mathbf{W}^i\mathbf{h}_{t-1}\right)
\end{aligned}
$$

where $g(\cdot)$ is a sigmoid function.

LSTMs

- Forget gates **f** decide what information to **forget** from the previous memory state.
- Input gates **i** decide which pieces of new information **g** to store in the current memory state
- Output gates **o** control which pieces of information in the current memory state to output.

Final points...

- You don't need to really understand the structure for LSTMs; just apply the model!
- We considered a dense "vanilla" recurrent layer. We can have recurrent convolutional layers instead (see e.g., `layer_conv_lstm_2d`), which captures temporal structure within sequences of images

Graph neural networks

Graphs

- A graph G can be defined as $G := (V, E)$ where V is a set of nodes/vertices (numbered 1 to d) and E is a collection of edges between nodes.
- We collect information about G in an **adjacency matrix** $A \in \mathbb{R}^{d \times d}$, which need not be symmetric.
- The (i, j)-th element of A is positive and non-zero if there is a connection between the i-th and j-th node. This could be equal to one, or a scalar denoting the "strength" of connection.
- We observe a vector of features $\mathbf{x}_i \in \mathbb{R}^p$ at each node and we can store this in a feature matrix $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_p)' \in \mathbb{R}^{d \times p}$.
- We might also have a vector of edge features as well, but for simplicity we will contain all edge information in A.

Example

Applications

Graphical data appears in a number of applications:

- Networks: social, traffic,...
- NLP;
- Fraud detection;
- Chemistry: molecular fingerprints;
- Environmental: spatial data as fully-connected graph.

Applications

Different types of graph learning:

- Node-level tasks;
- Graph-level tasks;
- Edge-level tasks.

What makes graph learning so hard?[12]

- The layers we have are for specific modality, e.g., vector for dense, image for CNN, sequence for RNN.
- The operations are defined for fixed dimension.
- Consider a CNN; this is for an image with pre-fixed dimension. Every pixel will always have the same number of neighbours!
- Graphs are often more complex, with variable size and different amounts of neighbours.
- Ordering in graphs does not matter. GNNs must be **permutationally-invariant**.

[12] Copyright belongs to `https://neptune.ai/blog/graph-neural-network-and-some-of-gnn-applications`

General structure of a GNN[13]

For node-level tasks – input a graph and output a graph.

At each hidden layer, update features at nodes via a form of **message passing** and agglomerate.

[13] From slides belonging to Thomas Kipf, University of Amsterdam

Convolutions on a grid[14]

- $\mathbf{h}_i \in \mathbb{R}^p$ is hidden feature at i-th pixel.
- Convolution involves adding up $\sum_i \mathbf{W}_i \mathbf{h}_i$.

The $(l+1)$-th hidden feature vector for the central pixel is

$$\mathbf{h}_4^{(l+1)} = g^{(l)}\left(\mathbf{W}_0^{(l)}\mathbf{h}_0^{(l)} + \mathbf{W}_1^{(l)}\mathbf{h}_1^{(l)} + \cdots + \mathbf{W}_8^{(l)}\mathbf{h}_8^{(l)}\right).$$

Not so helpful for graphs! What if we switch $\mathbf{h}_1$ and $\mathbf{h}_2$?

[14] From slides belonging to Thomas Kipf, University of Amsterdam

Convolutions on a graph[15]

The $(l+1)$-th hidden feature vector for the i-th node is

$$\mathbf{h}_i^{(l+1)} = g^{(l)}\left(\mathbf{h}_i^{(l)}\mathbf{W}_0^{(l)} + \sum_{j\in\mathcal{N}_i}\frac{1}{c_{ij}}\mathbf{h}_j^{(l)}\mathbf{W}_1^{(l)}\right):$$

$\mathcal{N}_i$ indexes the neighbours for node i and c_{ij} is some norming constant (fixed or trainable). Note that this update is **permutation-invariant**, as the weights are shared across locations.

[15] From slides belonging to Thomas Kipf, University of Amsterdam

With or without skip-connections?

We can go a step further and enforce that $\mathbf{W}_1^{(l)} = \mathbf{W}_0^{(l)}$. Then:

$$\mathbf{h}_i^{(l+1)} = g^{(l)}\left(\mathbf{h}_i^{(l)}\mathbf{W}_1^{(l)} + \sum_{j\in\mathcal{N}_i}\frac{1}{c_{ij}}\mathbf{h}_j^{(l)}\mathbf{W}_1^{(l)}\right).$$

Letting $\mathbf{W}_1^{(l)} \neq \mathbf{W}_0^{(l)}$ induces a **skip-connection**.

GCN: Matrix notation

Start with:

$$\mathbf{h}_i^{(l+1)} = g^{(l)}\left(\mathbf{h}_i^{(l)}\mathbf{W}_1^{(l)} + \sum_{j\in\mathcal{N}_i}\frac{1}{c_{ij}}\mathbf{h}_j^{(l)}\mathbf{W}_1^{(l)}\right).$$

Collect in $\mathbf{H}^{(l)} = (\mathbf{h}_1^{(l)}, \ldots, \mathbf{h}_d^{(l)})' \in \mathbb{R}^{d\times p_l}$ the hidden feature matrix, where p_l is the number of features in the l-th layer, with $\mathbf{H}^{(0)} = \mathbf{X}$.
Collect in $\mathbf{W}^{(l)} = (\mathbf{W}_1^{(l)}, \ldots, \mathbf{W}_1^{(l)})' \in \mathbb{R}^{p_l\times p_{l-1}}$ the weights.
Then $\mathbf{H}^{(l)} = \mathbf{g}^{(l)}(\tilde{A}\mathbf{H}^{(l-1)}\mathbf{W}^{(l)})$, for $\tilde{A} = A + I$.
What happens to c_{ij}?

GCN: Matrix notation

The matrix $\tilde{A}$ needs to be normalised! Or the hidden output at nodes with many neighbours will be much larger.
Solution: normalise $\tilde{A}$ by the degree matrix $\tilde{D}$. Then we have

$$\mathbf{H}^{(l)} = \mathbf{g}^{(l)}(\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}\mathbf{H}^{(l-1)}\mathbf{W}^{(l)}).$$

This is equivalent to having

$$\mathbf{h}_i^{(l+1)} = g^{(l)}\left(\frac{1}{|\mathcal{N}_i|}\mathbf{h}_i^{(l)}\mathbf{W}_1^{(l)} + \frac{1}{|\mathcal{N}_i|}\sum_{j\in\mathcal{N}_i}\mathbf{h}_j^{(l)}\mathbf{W}_1^{(l)}\right).$$

Message passing vs. convolutional vs. attentional

We have described the simplest graph convolutional neural network layer[16]. This is all we will use in the practicals!
The general structure of the **propagation** of information in a GNN layer can always be written as

$$\mathbf{h}_i^{(l)} = \phi^{(l)}\{\mathbf{h}_i^{(l-1)}, \{\mathbf{h}_j^{(l-1)}\}_{j\in\mathcal{N}_i}\},$$

where *propagation module* $\phi^{(l)}$ is learnable and permutation-invariant. Often of the form

$$\phi^{(l)}(\mathbf{x}, \mathbf{y}) = g^{(l)}(\mathbf{W}_0^{(l)}\mathbf{x} + \mathbf{W}_1^{(l)}\mathbf{y}),$$

for weights $\mathbf{W}_0^{(l)}, \mathbf{W}_1^{(l)}$.

[16] Kipf and Welling. Semi-supervised classification with graph convolutional networks. arXiv:1609.02907, 2016.

Convolutional $\subseteq$ attentional $\subseteq$ message-passing

Convolutional:

$$\mathbf{h}_i^{(l+1)} = \phi^{(l)}\left(\mathbf{h}_i^{(l)}, \bigoplus_{j\in\mathcal{N}_i} \frac{1}{c_{ij}}\psi\{\mathbf{h}_j^{(l)}\}\right).$$

Attentional:

$$\mathbf{h}_i^{(l+1)} = \phi^{(l)}\left(\mathbf{h}_i^{(l)}, \bigoplus_{j\in\mathcal{N}_i} \frac{1}{a(\mathbf{h}_i^{(l)}, \mathbf{h}_j^{(l)})}\psi\{\mathbf{h}_j^{(l)}\}\right),$$

with self-attention mechanism $a(\mathbf{h}_i^{(l)}, \mathbf{h}_j^{(l)})$.
Message-passing:

$$\mathbf{h}_i^{(l+1)} = \phi^{(l)}\left(\mathbf{h}_i^{(l)}, \bigoplus_{j\in\mathcal{N}_i} \psi\{(\mathbf{h}_i^{(l)}, \mathbf{h}_j^{(l)})\}\right),$$

where $\psi(\mathbf{x})$ is linear and $\bigoplus$ is a permutation-invariant aggregation function, e..g, sum, mean, max.

Some examples of propagation modules[17]

Let's have a look here
`https://graphneural.network/layers/convolution/`.

[17] Zhou, Cui, Hu, Zhang, Yang, Liu, Wang, Li, Sun. (2020). Graph neural networks: A review of methods and applications.

Sampling[18]

For very large graphs, we might not want to use all nodes in the neighbourhood $\mathcal{N}_i$. We can instead sample nodes.

[18] Zhou, Cui, Hu, Zhang, Yang, Liu, Wang, Li, Sun. (2020). Graph neural networks: A review of methods and applications.

Graph level prediction

So we can apply propagation modules throughout a number of hidden layers. But at the end we get a graph. What happens if we want to output a scaler or vector?

For CNNs, we did a form of dimensionality-reduction using **pooling** layers.

Pooling

Pooling for graphs can be done either locally or globally:

- **Local** pooling: outputs a coarser graph with less vertices;
- **Global** pooling: outputs a vector, rather than a graph.

Again, lots to choose from
`https://graphneural.network/layers/pooling/`

TopKPool[19]

For a ratio $k \in (0, 1)$; keep $\lfloor kd \rfloor$ nodes. For input graph $(\mathbf{H}, A)$ for $\mathbf{H} \in \mathbb{R}^{d \times p}$, $A \in \mathbb{R}^{d \times d}$, and learnable projection vector $\mathbf{p} \in \mathbb{R}^p$.

- Get scores $\mathbf{y} = \mathbf{H} \times \mathbf{p} / \|\mathbf{p}\|$.
- Keep indices of $\mathbf{H}$ and A that provide the top $\lfloor kd \rfloor$ scores; denote former by $\tilde{\mathbf{H}}$.
- Re-weight $\tilde{\mathbf{H}}$ by sigmoid of indexed $\mathbf{y}$ to ensure differentiability.

[19] Cangea, Cătălina, Petar Veličković, Nikola Jovanović, Thomas Kipf, and Pietro Liò. "Towards sparse hierarchical graph classifiers." arXiv preprint arXiv:1811.01287 (2018).

TopKPool[20]

[20] Gao and Ji. (2019). Graph U-Nets.

SortPool

Pipeline[21]

The general design pipeline for a GNN model.

[21] Zhou, Cui, Hu, Zhang, Yang, Liu, Wang, Li, Sun. (2020). Graph neural networks: A review of methods and applications.

Thanks for your attention!